A **Knowledge Transfer (KT) Session** is a structured process where information, skills, expertise, and best practices are shared between team members, departments, or organizations. The goal is to ensure **smooth continuity of work** and **effective knowledge retention**.

When is KT Needed?

A team member is **leaving or switching roles**
Onboarding **new employees or team members**
Transitioning a **project to another team**
Introducing **new tools, processes, or technology**
Cross-training within the team for **backup support**

Key Components of a KT Session

1 Overview of the Topic/Project (Objectives, Scope, Goals)
2 Key Processes & Workflows (How things are done)
3 Challenges & Solutions (Common issues and how to resolve them)
4 Tools & Resources (Software, Documents, Guidelines)
5 Q&A Session (Clarifications, Follow-ups)

Example of a KT Session:

Imagine a **Software Tester** is moving to a new team. Before leaving, they conduct a KT session to explain:

- **Testing Strategies** used in the project
- **Automation scripts** and how they work
- **Common bugs** found and their fixes
- **Test reports & documentation** for future reference

This ensures that the **new tester can continue the work efficiently** without starting from scratch.

The image provides an overview of SDLC (Software Development Life Cycle). Let's break it down with explanations and examples:

1. **Systematic Process**:
    - SDLC follows a structured approach to software development, ensuring each step is well-defined.
    - *Example:* A company developing a banking application follows SDLC phases like requirement gathering, design, development, testing, deployment, and maintenance.
2. **Iterative Process**:
    - SDLC often involves repeating cycles to improve the software through feedback and iterations.
    - *Example:* Agile methodology follows an iterative model where new features are added in multiple sprints.
3. **Detailed Planning (Develop, Fix, Maintain)**:
    - A clear roadmap for development, fixing issues, and maintaining software.
    - *Example:* A healthcare software company plans for feature updates and bug fixes post-release.
4. **Increases Development Speed**:
    - A structured SDLC ensures efficient coding, testing, and deployment.

- *Example:* DevOps integrates automation tools like Jenkins and Docker to speed up development.

5. **Improves Client Relations**:
   - Following SDLC ensures transparency and better communication with clients.
   - *Example:* A retail company requests a custom inventory management system, and the development team provides regular progress updates.
6. **Reduces Risks**:
   - SDLC includes risk analysis, reducing project failures.
   - *Example:* Before launching an e-commerce platform, security risks like payment gateway vulnerabilities are analyzed.
7. **Helps in Planning, Scheduling, Estimating**:
   - SDLC helps in project management by defining timelines and resource allocation.
   - *Example:* A gaming company estimates a 12-month development timeline with 50 developers and designers.
8. **Standard Activities (Meetings)**:
   - Regular meetings for progress tracking, requirement changes, and issue resolution.
   - *Example:* A software company conducts daily standup meetings in Scrum methodology.

The image highlights key characteristics of a well-structured **Software Development Life Cycle (SDLC)**. Let’s break them down with explanations and examples:

1. Clearly-Defined

- The development process should follow a structured and well-documented methodology.
- *Example:* A **banking application** development follows predefined SDLC phases—Requirement Analysis, Design, Development, Testing, Deployment, and Maintenance—to ensure smooth execution.

2. High Quality

- The software must meet industry standards and function as expected without defects.
- *Example:* A **healthcare system** must go through rigorous testing to ensure there are no errors in patient data processing, as mistakes can have serious consequences.

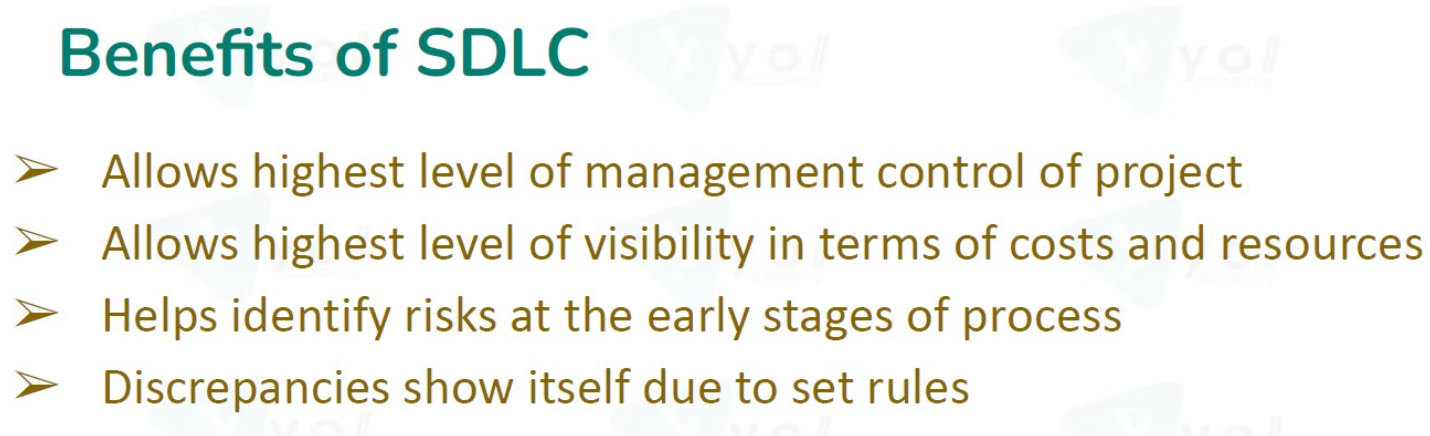
3. Cost-Effective

- The development process should optimize resources and minimize waste while staying within budget.
- *Example:* A startup building an **e-commerce website** chooses **open-source technologies** like Magento or WooCommerce instead of expensive proprietary solutions to reduce costs.

4. Exceed Customer Expectations

- The software should not just meet but surpass customer needs in usability, performance, and features.
- *Example:* A **ride-hailing app** (like Uber) introduces AI-based estimated arrival times, making the experience smoother for users beyond just basic cab booking.

A well-planned SDLC ensures the development process is efficient, high-quality, cost-effective, and customer-centric.


Benefits of SDLC
Allows highest level of management control of project
Allows highest level of visibility in terms of costs and resources
Helps identify risks at the early stages of process
Discrepancies show itself due to set rules

The image outlines the **Benefits of SDLC** (Software Development Life Cycle). Let’s break them down with detailed explanations and real-world examples.

1. Allows Highest Level of Management Control of Project

- SDLC provides a structured framework where managers can oversee the project efficiently.
- *Example:* In a **banking software development project**, managers monitor each phase (requirement gathering, design, coding, testing, deployment) to ensure everything aligns with business goals and regulatory standards.

2. Allows Highest Level of Visibility in Terms of Costs and Resources

- SDLC provides a clear roadmap of expenses, time, and workforce needed at each stage.
- *Example:* A **healthcare software company** developing an **Electronic Medical Records (EMR) system** can predict infrastructure costs (servers, cloud storage) and resource allocation (developers, testers) before starting, preventing unexpected budget overruns.

3. Helps Identify Risks at the Early Stages of Process

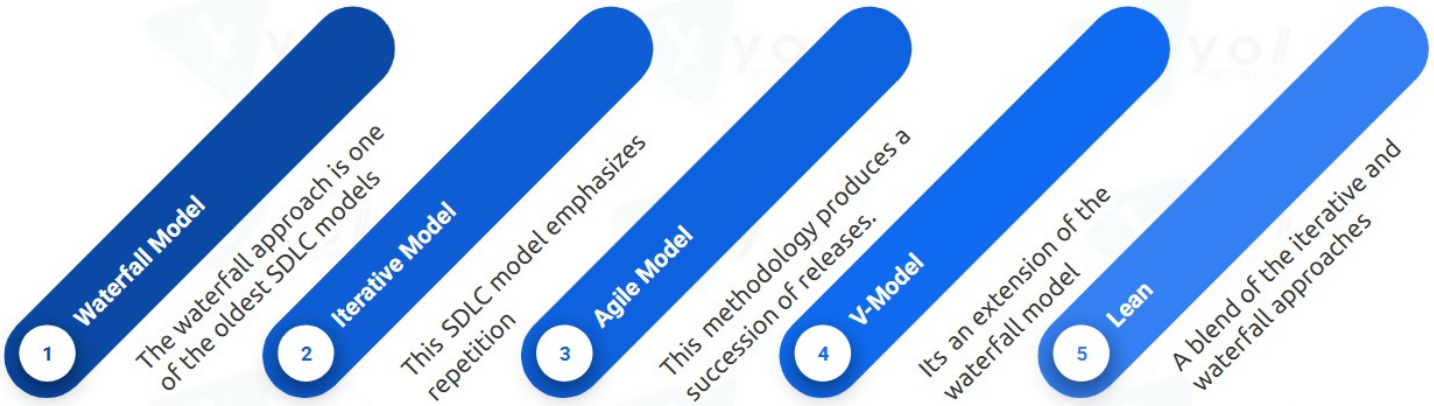
- By analyzing potential risks early, businesses can avoid costly failures.
- *Example:* A **financial transactions platform** like PayPal integrates **security risk analysis** early in SDLC to prevent fraud, ensuring strong encryption and compliance with global payment security standards.

4. Discrepancies (Contradictions) Show Itself Due to Set Rules

- SDLC enforces strict guidelines, making it easier to detect issues before deployment.
- *Example:* In an **e-commerce website development**, if payment processing fails during testing, developers can correct it before release, preventing financial losses and customer dissatisfaction.

By following SDLC, organizations gain better control, cost predictability, risk mitigation, and error detection, leading to successful project execution.

SDLC Models



1. Waterfall Model:

- **What it is**: A linear and sequential approach where each phase of the SDLC (e.g., requirements, design, development, testing) is completed before moving to the next.
- **Example**: Imagine building a **simple calculator app**.
  - **Step 1**: Gather all requirements (e.g., the app should perform addition, subtraction, multiplication, and division).
  - **Step 2**: Design the app (e.g., create wireframes and technical specifications).
  - **Step 3**: Develop the app (e.g., write the code for each function).
  - **Step 4**: Test the app (e.g., ensure all functions work correctly).
  - **Step 5**: Deploy the app to users.
- **When to use**: For small, well-defined projects where requirements are unlikely to change.

---

2. Agile Model:

- **What it is**: An iterative and flexible approach where the project is broken into small increments or "sprints." Each sprint delivers a working piece of software.
- **Example**: Imagine building a **social media app**.
  - **Sprint 1**: Develop and release a basic version with user registration and login.

    - **Sprint 2**: Add a feature for posting and sharing updates.
    - **Sprint 3**: Add a messaging feature.
    - **Sprint 4**: Add a photo-sharing feature.
- **When to use**: For projects where requirements are expected to change or evolve over time.

---

3. Iterative Model:

- **What it is**: A model where the software is developed in cycles (iterations). Each iteration produces a working version of the software, which is improved upon in subsequent iterations.
- **Example**: Imagine building an **online shopping website**.
    - **Iteration 1**: Develop a basic version with product listings and a shopping cart.
    - **Iteration 2**: Add a payment gateway for checkout.
    - **Iteration 3**: Add user reviews and ratings.
    - **Iteration 4**: Add a recommendation system.
- **When to use**: For large projects where you want to deliver a working product early and improve it over time.

---

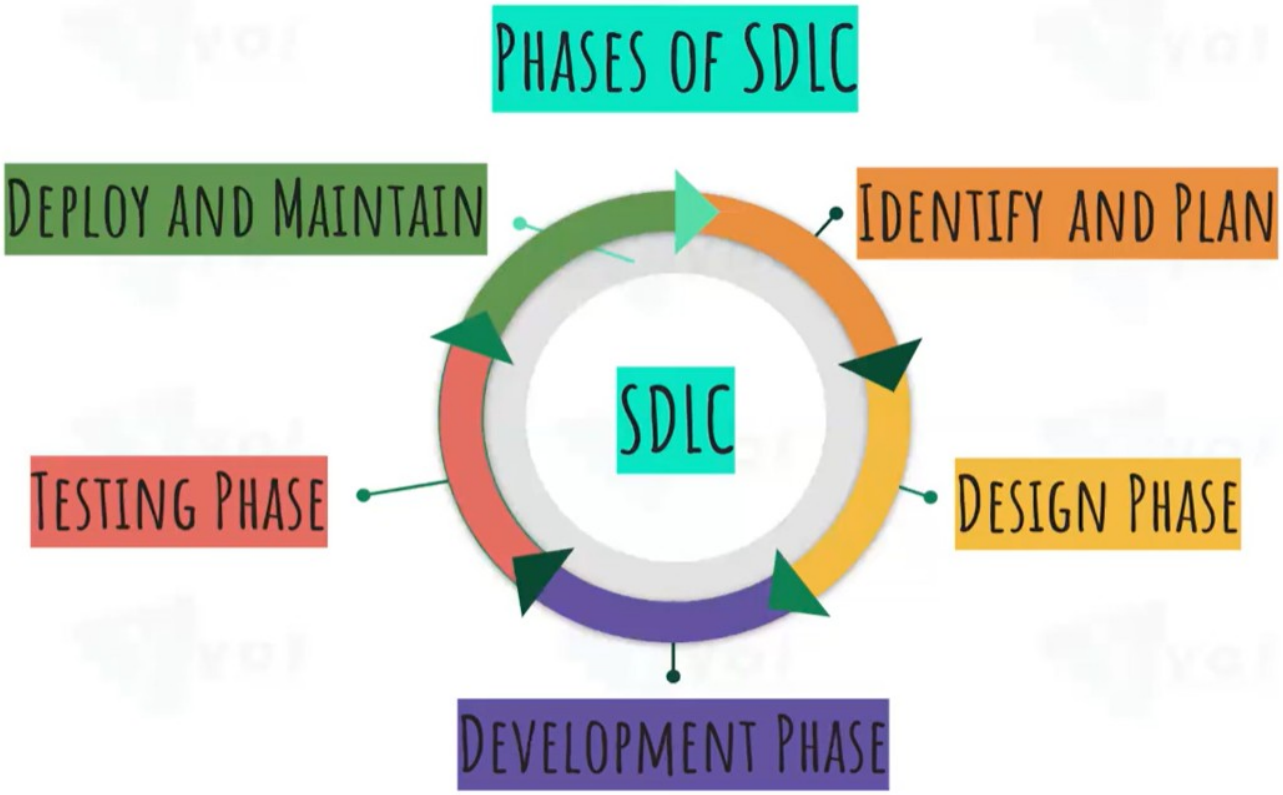
4. Spiral Model:

- **What it is**: A risk-driven model that combines elements of the Waterfall and Iterative models. It involves repeated cycles (spirals) of planning, risk analysis, development, and evaluation.
- **Example**: Imagine building a **banking app**.
    - **Spiral 1**: Identify risks (e.g., security vulnerabilities) and develop a basic version with login and account balance features.
    - **Spiral 2**: Address security risks and add features like money transfer and bill payments.
    - **Spiral 3**: Add advanced features like investment tracking and loan applications.
- **When to use**: For complex, high-risk projects where risk management is critical.

Summary with Examples:

1. **Waterfall Model**: A linear approach (e.g., building a calculator app step by step).
2. **Agile Model**: An iterative and flexible approach (e.g., building a social media app in sprints).
3. **Iterative Model**: A cyclical approach (e.g., building an online shopping website in iterations).
4. **Spiral Model**: A risk-driven approach (e.g., building a banking app with risk analysis in each cycle).

Each SDLC model has its strengths and is suited for different types of projects. Choosing the right model depends on the project's complexity, requirements, and risk factors.



The **Phases of the Software Development Life Cycle (SDLC)** using a circular flow diagram. SDLC is a systematic approach used in software engineering to design, develop, test, and maintain software efficiently.

. Identify and Plan (Requirement Analysis):

- **What it is**: This is the first phase where you understand the needs of stakeholders and define the project goals.
- **Activities**:
  - Talk to stakeholders (e.g., clients, users) to understand their requirements.
  - Define the scope, goals, and feasibility of the project.
  - Create a project plan, including timelines and resources.

- **Example**: Imagine you're building a **mobile app for online food delivery**.
  - Stakeholders might want features like **menu browsing**, **order placement**, and **payment processing**.
  - You document these requirements and create a plan for how to build the app.

---

2. Design Phase:

- **What it is**: In this phase, you create the blueprint for the software based on the requirements.
- **Activities**:
  - Design the system architecture (e.g., how the app will be structured).
  - Plan the user interface (UI) and user experience (UX).
  - Define technical specifications, such as the database structure and APIs.
- **Example**: For the **food delivery app**:
  - You design how the app will look (e.g., screens for browsing menus, placing orders).
  - You plan the database to store user information, menus, and orders.
  - You decide on the technologies to use (e.g., React Native for the app, MongoDB for the database).

---

3. Development Phase:

- **What it is**: This is where the actual coding and programming happen.
- **Activities**:
  - Developers write code based on the design specifications.
  - Implement features like user registration, menu browsing, and payment processing.
  - Use version control (e.g., Git) to manage code changes and document the process.
- **Example**: For the **food delivery app**:
  - Developers write code for the app's features"aero"") and backend (e.g., handling orders and payments).
  - They ensure the app connects to the database and APIs as planned.

---

4. Testing Phase:

- **What it is**: In this phase, the software is tested to ensure it works as expected and is free of bugs.
- **Activities**:
  - Testers check for bugs and errors in the software.
  - Perform functional testing (e.g., does the order placement work?).
  - Conduct performance testing (e.g., can the app handle 1,000 users at once?).
- **Example**: For the **food delivery app**:
  - Testers check if users can browse menus, place orders, and make payments without errors.

    - They also test if the app works smoothly on different devices (e.g., iOS, Android).

---

5. Deploy and Maintain:

- **What it is**: Once the software is tested and approved, it is released to users. After release, it requires regular maintenance.
- **Activities**:
    - Deploy the software to production (e.g., release the app on the App Store and Google Play).
    - Monitor the software for issues and fix bugs reported by users.
    - Release updates to add new features or improve performance.
- **Example**: For the **food delivery app**:
    - The app is released to the public, and users start downloading and using it.
    - Developers fix bugs (e.g., a payment issue) and release updates (e.g., adding a new feature like live order tracking).

Circular Flow of SDLC:

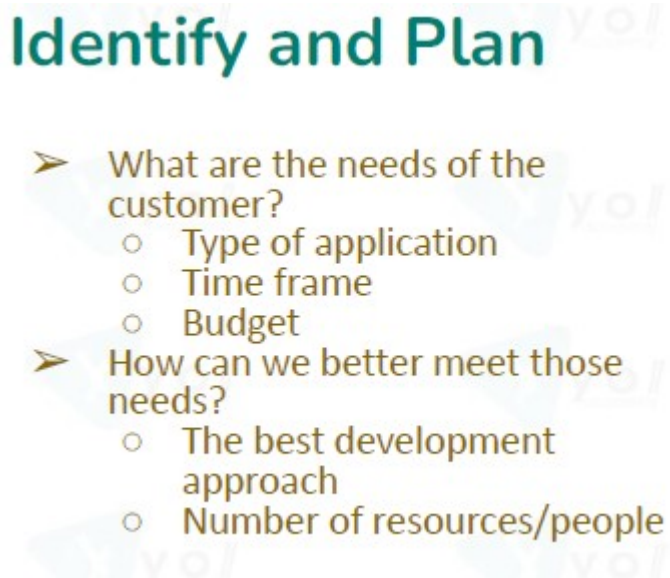
The SDLC is a **continuous cycle**. After the **Deploy and Maintain** phase, the process often loops back to the **Identify and Plan** phase for the next version or update of the software. For example:

- After releasing the food delivery app, stakeholders might request new features (e.g., **loyalty rewards** or **group ordering**).
- The cycle starts again: **Plan → Design → Develop → Test → Deploy**.

Summary with Examples:

1. **Identify and Plan**: Understand what the food delivery app should do (e.g., order placement, payment).
2. **Design Phase**: Create the app’s blueprint (e.g., UI design, database structure).
3. **Development Phase**: Write code to build the app (e.g., coding the menu browsing feature).
4. **Testing Phase**: Test the app to ensure it works (e.g., check if payments process correctly).
5. **Deploy and Maintain**: Release the app and fix issues (e.g., update the app to fix bugs).

The **circular flow** ensures the software evolves over time, meeting user needs and staying functional. Each phase builds on the previous one, creating a systematic and efficient process for software development.



the **"Identify and Plan"** phase of the **Software Development Life Cycle (SDLC)**. This phase is crucial as it lays the foundation for the entire software development process.

Key Aspects Explained:

1. **Understanding Customer Needs:**
    - **Type of Application:** Determine what kind of software is required (e.g., web, mobile, desktop, enterprise solution).
    - **Time Frame:** Define the project deadline and delivery schedule.
    - **Budget:** Assess financial constraints to ensure cost-effective development.
2. **Planning to Meet Customer Needs:**
    - **Best Development Approach:** Decide whether to use Agile, Waterfall, or another methodology.
    - **Number of Resources/People:** Identify the team size and skill sets required for the project.

This phase ensures that all stakeholders are aligned on project expectations, reducing risks and improving efficiency in later stages.

the **"Design"** phase of the **Software Development Life Cycle (SDLC)**. This phase focuses on defining the **architecture and blueprint** of the software before development begins.

Key Aspects Explained:

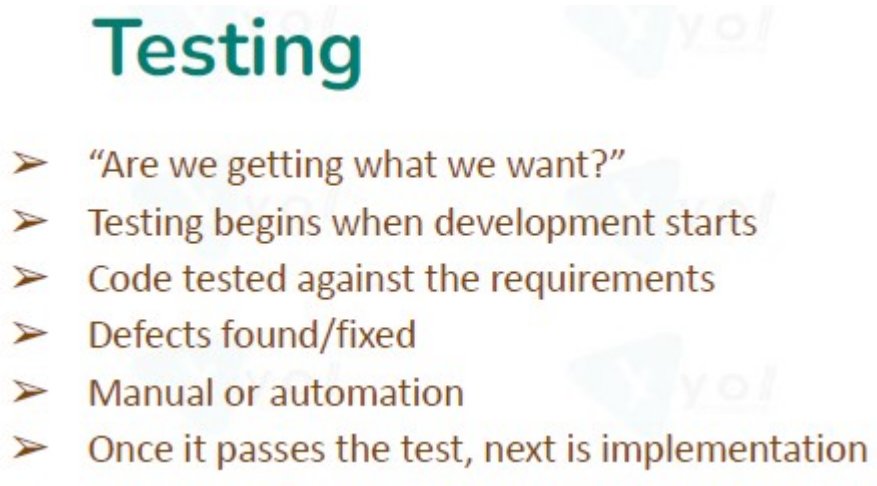
1. **"How will we get what we want?"**
    - This refers to translating requirements into a structured plan.
2. **Main Player is the Architect**
    - Software architects are responsible for designing the system structure, database, and overall technical flow.
3. **More Than One Design Plans**
    - Multiple design models may be created to explore different approaches.
4. **Most Popular One Gets Developed**
    - The best or most feasible design is selected for implementation.

This phase ensures that the software is built on a solid foundation with well-planned components and interactions, reducing future risks and inefficiencies.

The **Development** phase is where the actual construction of the project takes place. Here’s another way to look at it:

1. **Writing the Code**
    - Developers start coding based on the previously designed architecture and specifications.
    - Each feature and functionality is implemented using programming languages and frameworks.
2. **Building the Application**
    - The individual components (such as front-end, back-end, and database) are developed and integrated to form a complete system.

3. **Choosing the Right Technology**
    - The programming languages and tools best suited for the project are selected (e.g., JavaScript for front-end, Python for back-end).
4. **A Relatively Straightforward Phase**
    - Compared to planning, designing, and testing, development is often seen as simpler because the requirements are already clear.
    - However, challenges like debugging and optimizing performance may arise.

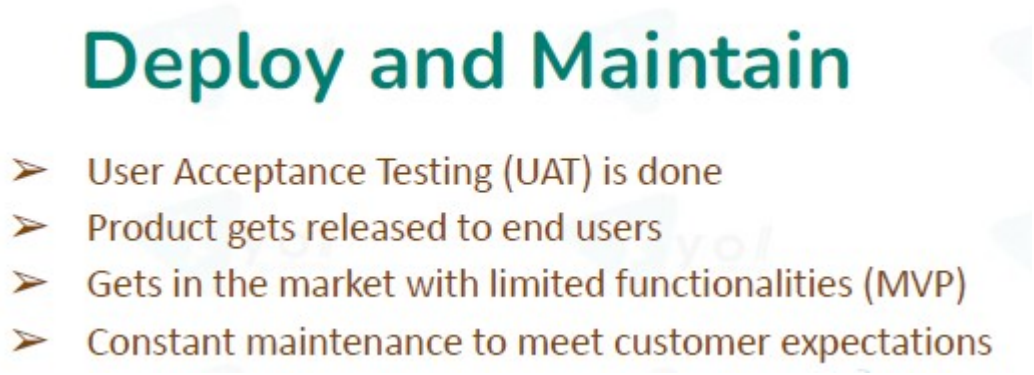

Testing
"Are we getting what we want?"
Testing begins when development starts
Code tested against the requirements
Defects found/fixed
Manual or automation
Once it passes the test, next is implementation

the **Testing** phase in a project or software development lifecycle. Here's a breakdown of the key points:

1. **"Are we getting what we want?"**
    - Testing ensures that the software meets the expected requirements and functions as intended.
    - It helps validate that the application behaves correctly under different conditions.
2. **Testing Begins When Development Starts**
    - In modern development practices (like Agile), testing is done alongside development to catch issues early.
    - This helps in reducing time and effort spent on fixing bugs later.
3. **Code Tested Against the Requirements**
    - Testers verify if the software meets the specified requirements.
    - Functional, performance, security, and usability testing are conducted.
4. **Defects Found/Fixed**
    - Bugs and issues are identified during testing and reported to developers for fixes.

- The fixed code is then retested to ensure the issue is resolved.

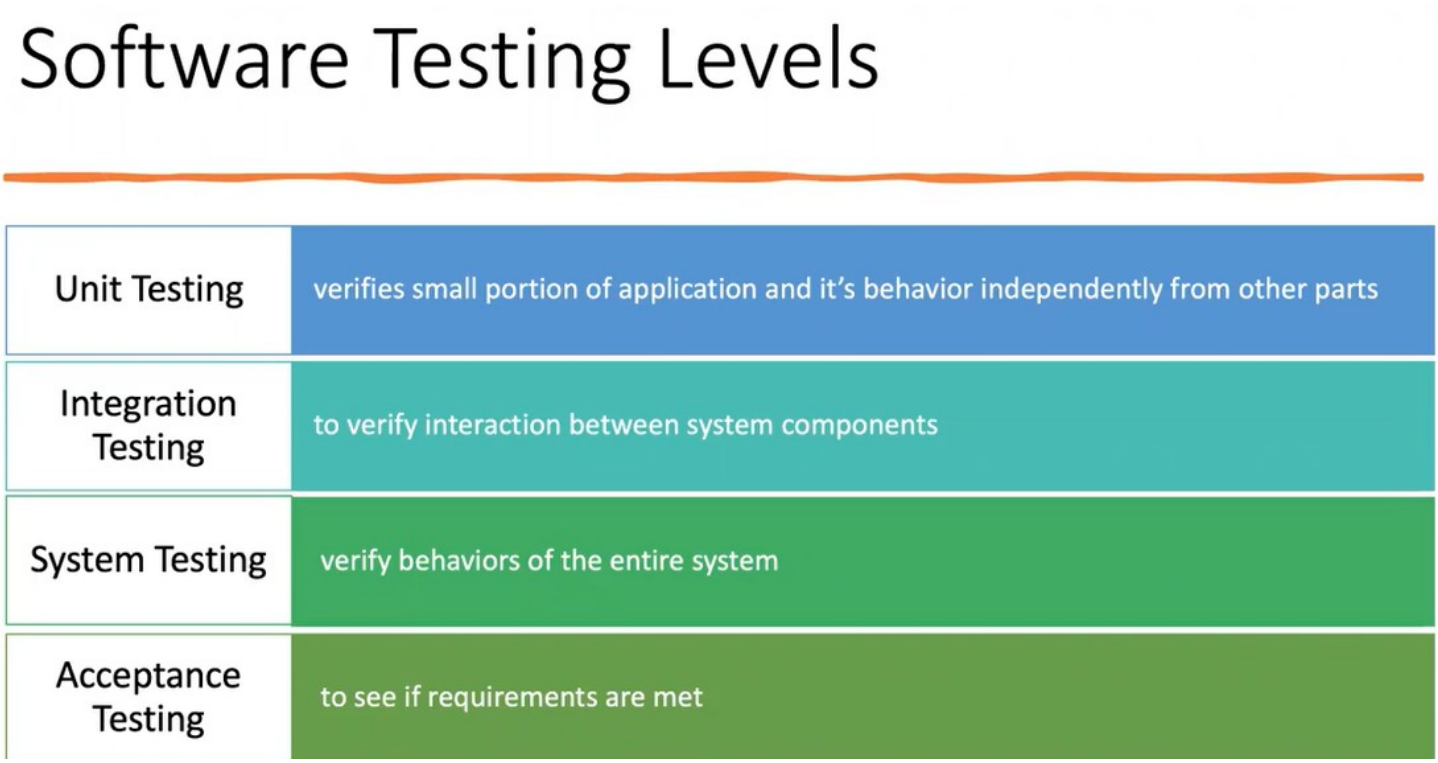
5. **Manual or Automation Testing**
   - Manual testing involves human testers executing test cases.
   - Automated testing uses scripts and tools (like Selenium, JUnit) to perform repetitive tests efficiently.
6. **Once It Passes the Test, Next Is Implementation**
   - If the software passes all tests successfully, it moves to the implementation or deployment phase.
   - This ensures that only stable and reliable software reaches end users.


Deploy and Maintain
User Acceptance Testing (UAT) is done
Product gets released to end users
Gets in the market with limited functionalities (MVP)
Constant maintenance to meet customer expectations

e **Deploy and Maintain** phase in a project lifecycle. Here's an explanation of each point:

1. **User Acceptance Testing (UAT) is Done**
   - Before releasing the product, it is tested by end-users or stakeholders to ensure it meets their expectations and works as intended.
   - If issues are found, they are fixed before deployment.
2. **Product Gets Released to End Users**
   - Once testing is complete, the software is deployed for real-world use.
   - This can be done in phases (gradual rollout) or all at once (full release).
3. **Gets in the Market with Limited Functionalities (MVP)**
   - The initial release might be a **Minimum Viable Product (MVP)** with essential features.
   - Additional features and improvements are added over time based on user feedback.
4. **Constant Maintenance to Meet Customer Expectations**
   - After deployment, regular updates, bug fixes, and performance enhancements are needed.

- Customer support and feedback play a crucial role in improving the product over time.


Software Testing Levels
Unit Testing
verifies small portion of application and it's behavior independently from other parts
Integration Testing
to verify interaction between system components
System Testing
verify behaviors of the entire system
Acceptance Testing
to see if requirements are met

1. **Unit Testing**: Imagine you're building a toy car. Unit testing is like checking each small part (like the wheels or the steering) to make sure each one works perfectly on its own. It involves verifying the functionality of individual components or units of the software in isolation from the rest of the system. The goal is to ensure that each small part of the application behaves as expected.

   Example: Imagine you're building a **calculator app**.
   1. **Unit Test**: You test the addition function separately to ensure it correctly adds two numbers.
      1. Input: 2 + 3
      2. Expected Output: 5
   2. You don't test the entire calculator at this stage—just the addition function to make sure it works perfectly on its own.

   Real-world analogy: If you're building a toy car, unit testing is like checking if the **wheels spin correctly** or if the **steering mechanism works** before assembling the car.

2. **Integration Testing**: Now, you want to see if all those small parts work well together. For example, do the wheels fit properly with the body? Does the steering connect correctly? This is integration testing—making sure all the parts work together as a team. After unit testing, integration testing checks how different units or components of the system work together. This level of testing aims to identify issues in the interaction between integrated components, such as data exchange and communication.

Example: In the **calculator app**, after testing the addition, subtraction, multiplication, and division functions individually (unit testing), you now test how they work together.

1. **Integration Test**: You test if the calculator can handle a sequence of operations, like 2 + 3 * 4.
    1. Expected Output: 14 (if it follows the correct order of operations).
2. Here, you're checking if the addition and multiplication functions interact correctly.

Real-world analogy: For the toy car, integration testing is like checking if the **wheels fit properly with the body** and if the **steering connects correctly** to the wheels. You're making sure all the parts work together as a team.

3. **System Testing**: Once the toy car is fully assembled, you test the whole car to see if it drives smoothly, turns correctly, and does everything it's supposed to do. This is system testing—checking the entire toy car to make sure it works as expected. his level involves testing the complete and integrated software system as a whole. The objective is to evaluate the system's compliance with the specified requirements and to ensure that it behaves correctly in various scenarios.

   Example: In the **calculator app**, system testing involves testing the entire application to see if it works as expected in all scenarios.

   1. **System Test**: You test the calculator with various inputs, like:
       1. 2 + 3 = 5
       2. 10 / 2 = 5
       3. 5 * 0 = 0
       4. 100 - 50 = 50
   2. You also test edge cases, like dividing by zero, to ensure the system handles errors gracefully.

   Real-world analogy: For the toy car, system testing is like assembling the car and testing if it **drives smoothly**, **turns correctly**, and **stops when needed**. You're checking if the whole car works as expected.

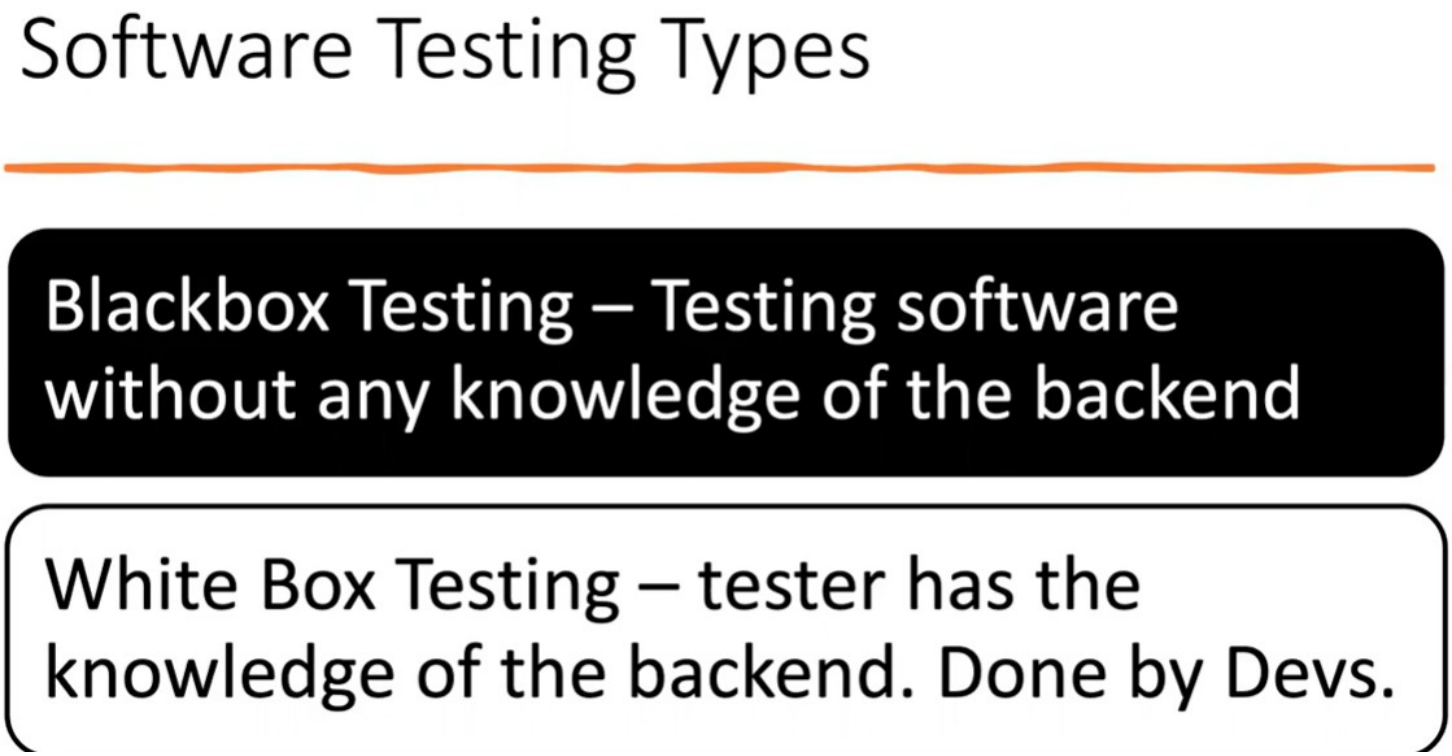
4. **Acceptance Testing**: Finally, you show the toy car to your friends or family to see if they like it and if it meets their expectations. This is acceptance testing—making sure the final product is ready and acceptable for use. This is the final level of testing, conducted to determine whether the software is ready for delivery. It involves verifying that the software meets the business requirements and is acceptable for deployment. This testing is often done with the involvement of stakeholders or end-users.

   Example: In the **calculator app**, acceptance testing involves showing the app to the client or end-users to see if it meets their expectations.

   1. **Acceptance Test**: The client might test the app by performing real-world calculations, like:
       1. Calculating a restaurant bill with tax and tip.
       2. Checking if the app is user-friendly and meets their needs.
   2. If the app works as expected and the client is satisfied, it's ready for delivery.

Real-world analogy: For the toy car, acceptance testing is like showing the car to your friends or family to see if they like it and if it meets their expectations. If they’re happy, the toy car is ready to be used!

Each step helps make sure the toy car (or software) is built correctly and works well from the smallest part to the whole thing!


Software Testing Types
Blackbox Testing – Testing software without any knowledge of the backend
White Box Testing – tester has the knowledge of the backend. Done by Devs.

1. Blackbox Testing:

- **What it is**: Testing the software without knowing how it works internally. Testers focus on the functionality and user experience.
- **How it's done**: Testers provide inputs and check if the outputs match the expected results, without looking at the code.
- **Purpose**: To ensure the software meets user requirements and behaves correctly in real-world scenarios.

Example:

- Imagine you're testing a **login page** of a website.
  - **Input**: You enter a username and password.
  - **Expected Output**: The system should log you in if the credentials are correct, or show an error message if they are wrong.
  - **Blackbox Testing**: You don't know how the system checks the username and password (e.g., whether it uses a database or an API). You only care if the login feature works as expected from the user's perspective.

Another example:

- Testing a **calculator app**:
  - **Input**: You enter `2 + 2`.
  - **Expected Output**: The app should display `4`.
  - **Blackbox Testing**: You don't care how the app calculates the result (e.g., whether it uses addition logic or a complex algorithm). You only care if the result is correct.

---

2. White Box Testing:

- **What it is**: Testing the software with full knowledge of its internal code, structure, and logic. This is usually done by developers.
- **How it's done**: Testers examine the code, check conditions, loops, and logic paths to ensure everything works as intended.
- **Purpose**: To verify the internal logic and structure of the code, ensuring all parts of the code are tested.

Example:

- Imagine you're testing a **function that calculates discounts** in an e-commerce app.
  - **Code Logic**: The function applies a 10% discount if the total purchase is over $100.
  - **White Box Testing**: You look at the code and test different scenarios:
    - If the total is $90, does the function skip the discount?
    - If the total is $110, does the function correctly apply a 10% discount?
    - Does the function handle edge cases, like negative values or zero?

Another example:

- Testing a **sorting algorithm**:
  - **Code Logic**: The algorithm sorts a list of numbers in ascending order.
  - **White Box Testing**: You examine the code and test:
    - Does the algorithm handle an empty list?
    - Does it sort a list with duplicate numbers correctly?
    - Does it handle a list with negative numbers?

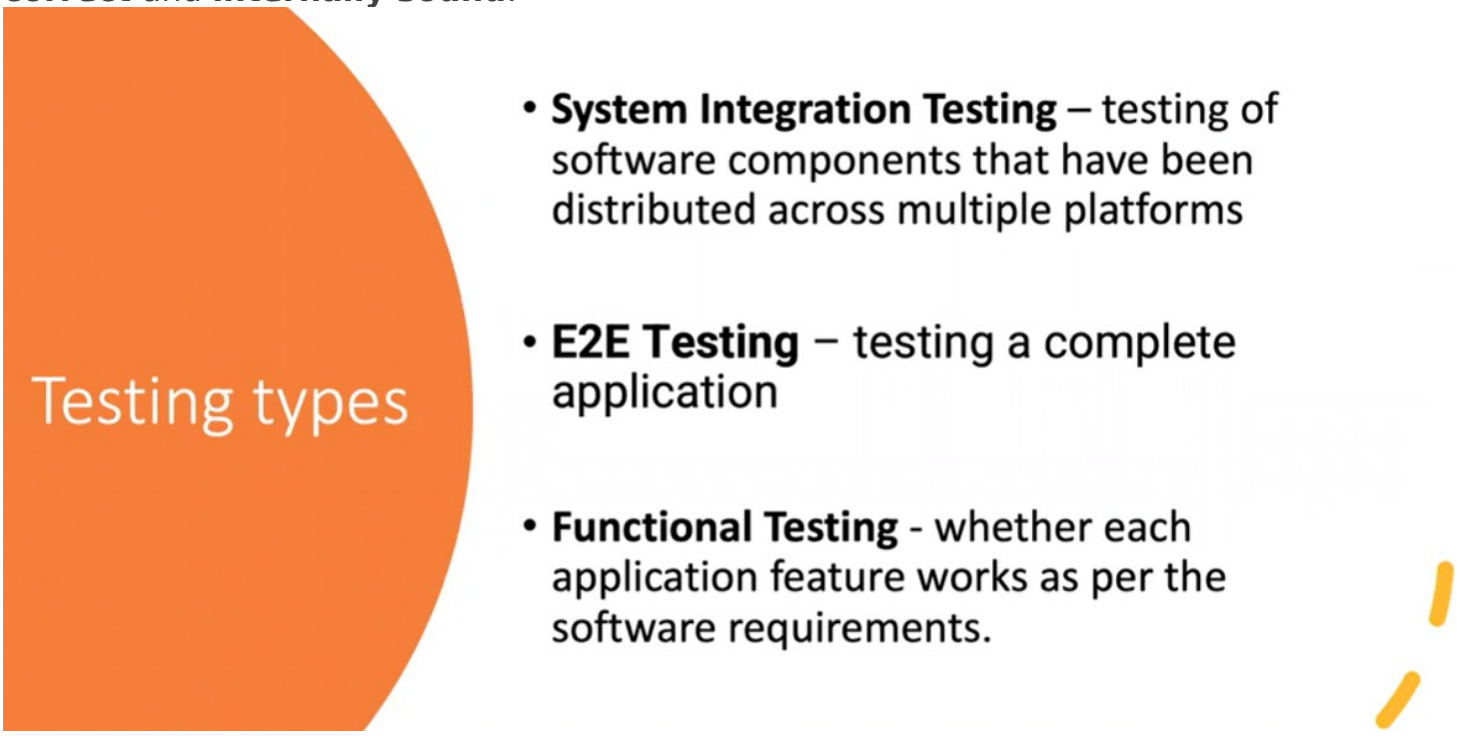
Summary:

- **Blackbox Testing**: Focuses on **what the software does** (functionality) without knowing **how it works internally**.
  - Example: Testing a login page or a calculator app from the user's perspective.
- **White Box Testing**: Focuses on **how the software works internally** (code, logic, and structure).
  - Example: Testing a discount calculation function or a sorting algorithm by examining the code.

Both types of testing are essential:

- **Blackbox Testing** ensures the software meets user requirements and behaves correctly in real-world scenarios.
- **White Box Testing** ensures the code is robust, efficient, and free of logical errors.

By combining both, you can ensure the software is both **functionally correct** and **internally sound**.


Testing types
• System Integration Testing – testing of software components that have been distributed across multiple platforms
• E2E Testing – testing a complete application
• Functional Testing - whether each application feature works as per the software requirements.

The document describes different types of software testing, each focusing on a specific aspect of the software to ensure it works correctly. Here's a simple explanation:

1. **System Integration Testing**:
    - **What it is**: This type of testing checks how different parts of the software, which might be on different platforms or systems, work together.
    - **Purpose**: To make sure all the different components can communicate and function properly as a unified system.
    - **Example**: Imagine you have an online shopping app. The app might use different services like a payment gateway, a product catalog, and a shipping service, each running on different servers or platforms. System Integration Testing would check if these services work together correctly. For instance, when you place an order, it should correctly process the payment, update the inventory, and schedule the shipment.
2. **End-to-End (E2E) Testing**:
    - **What it is**: This involves testing the entire application from start to finish, just like a real user would.
    - **Purpose**: To ensure that the whole application works correctly in a complete workflow, checking that all parts are connected and function as expected.

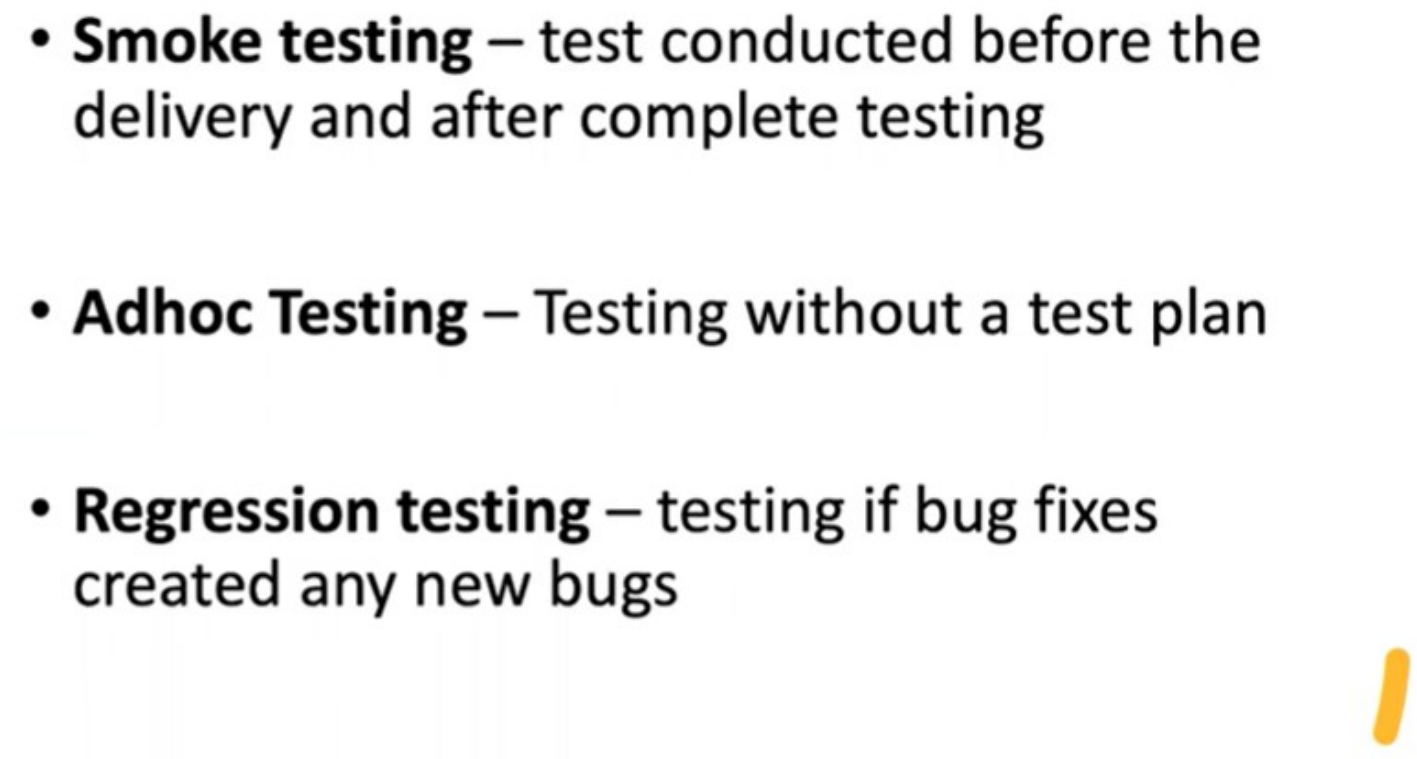
    - **Example**: Using the same online shopping app, End-to-End Testing would simulate a complete user journey. This could include steps like searching for a product, adding it to the cart, entering payment details, and completing the purchase. The test would check if the entire process works smoothly from start to finish.

3. **Functional Testing**:
    - **What it is**: This type of testing focuses on checking each feature of the application to see if it works according to the requirements.
    - **Purpose**: To verify that every function of the software does what it's supposed to do, based on the design and user needs.
4. **Example**: In the online shopping app, Functional Testing would focus on individual features. For example, you would test if the search bar correctly finds products when you type in keywords, or if the "Add to Cart" button properly adds items to your cart. Each feature is tested to ensure it works according to the requirements.
    -

In summary:

- **System Integration Testing** ensures different parts of the system work well together.
- **End-to-End Testing** checks the entire application to make sure everything works from start to finish.
- **Functional Testing** makes sure each feature of the application works correctly.

Each type of testing is important to ensure the software is reliable and meets the needs of its users.


• Smoke testing – test conducted before the delivery and after complete testing
• Adhoc Testing – Testing without a test plan
• Regression testing – testing if bug fixes created any new bugs

4. Smoke Testing:

- **What it is**: A quick, preliminary test conducted to check if the basic functionalities of the software are working after a build or before delivery. It ensures that the software is stable enough for further testing.
- **Example**: Imagine you've built a **new version of a mobile banking app**.
    - **Smoke Test**: You quickly test the core features, like logging in, checking the account balance, and transferring money, to ensure they work.
    - If these basic features fail, the build is rejected, and further testing is halted until the issues are fixed.
- **Purpose**: To catch major issues early and avoid wasting time on detailed testing if the build is unstable.

---

5. Adhoc Testing:

- **What it is**: Testing performed without a formal test plan or documentation. It is informal and often done on the fly to find defects that might not be caught by structured testing.
- **Example**: Continuing with the **mobile banking app**:
    - **Adhoc Test**: A tester randomly tries to transfer a negative amount of money or logs in with an incorrect password multiple times to see how the app behaves.

    - The goal is to find unexpected issues or edge cases that weren't covered in the formal test plan.
- **Purpose**: To uncover hidden bugs by exploring the software in an unstructured way.

---

6. Regression Testing:

- **What it is**: Testing performed to ensure that recent changes (like bug fixes or new features) haven't introduced new bugs or broken existing functionality.
- **Example**: In the **mobile banking app**, suppose a bug was found in the money transfer feature, and developers fixed it.
    - **Regression Test**: Testers re-run tests on the money transfer feature to ensure the bug is fixed. They also test other related features (e.g., account balance updates, transaction history) to ensure the fix didn't break anything else.
- **Purpose**: To ensure that the software remains stable and functional after changes are made.

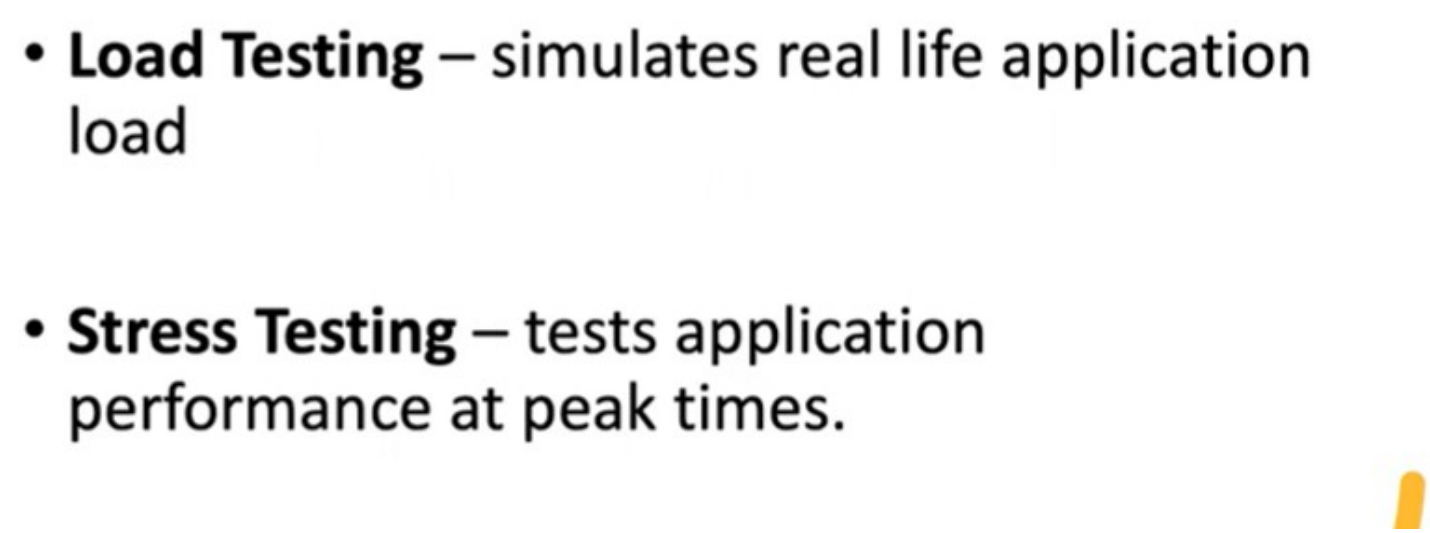
Summary with Examples:

Smoke Testing: A quick check to ensure the basic functionalities of the software work (e.g., testing login, balance check, and money transfer in a banking app).

Adhoc Testing: Informal testing without a plan to find unexpected issues (e.g., trying to transfer a negative amount in a banking app).

Regression Testing: Re-testing to ensure new changes haven't broken existing functionality (e.g., re-testing the money transfer feature after a bug fix).

Each type of testing plays a crucial role in ensuring the software is stable, functional, and free of defects.



7. Load Testing:

- **What it is**: Load testing simulates real-life usage of the software to see how it performs under expected user loads. The goal is to ensure the application can handle the expected number of users or transactions without issues.
- **Example**: Imagine you're testing an **e-commerce website** like Amazon.
  - **Load Test**: You simulate 10,000 users browsing products, adding items to their cart, and checking out simultaneously.
  - You monitor the website's performance (e.g., response time, server load) to ensure it can handle the expected traffic without slowing down or crashing.
- **Purpose**: To ensure the application performs well under normal or expected conditions.

---

8. Stress Testing:

- **What it is**: Stress testing pushes the software beyond its normal operating limits to see how it performs under extreme conditions, such as peak traffic or resource exhaustion. The goal is to identify the breaking point of the application.
- **Example**: Continuing with the **e-commerce website**:
  - **Stress Test**: You simulate 50,000 users trying to access the website during a flash sale or Black Friday event.
  - You monitor how the website behaves under this extreme load—does it crash, slow down significantly, or handle the load gracefully?
- **Purpose**: To identify the application's limits and ensure it can recover from failure (e.g., by scaling resources or showing a friendly error message).

Key Differences:

- **Load Testing**: Focuses on **expected usage** (e.g., 10,000 users on an e-commerce site).
- **Stress Testing**: Focuses on **extreme conditions** (e.g., 50,000 users during a flash sale).

Real-World Analogy:

- **Load Testing**: Like testing a bridge to see if it can handle the usual daily traffic of 1,000 cars.
- **Stress Testing**: Like testing the same bridge to see if it can handle 5,000 cars during a festival or emergency situation.

Summary with Examples:

1. **Load Testing**: Simulates real-life usage to ensure the application performs well under expected conditions (e.g., 10,000 users on an e-commerce site).
2. **Stress Testing**: Tests the application's performance under extreme conditions to identify its breaking point (e.g., 50,000 users during a flash sale).

Both types of testing are crucial for ensuring the software is reliable, scalable, and can handle real-world usage scenarios.

Agile
Opposite of the Waterfall Model
Most popular model
Fast paced and flexible
Enables all phases for everybody
Work is broken into small iterations (2-4 weeks called "Sprints")
AGILE

The **Agile model** is a popular and flexible approach to software development. Let's break it down with **examples** to make it easier to understand:

Key Features of Agile:

1. **Opposite of the Waterfall Model**:
    - Unlike the Waterfall model, which is linear and rigid, Agile is iterative and flexible. It allows for changes and improvements throughout the development process.
    - **Example**: In the Waterfall model, you'd complete the entire design of a **mobile app** before starting development. In Agile, you design and develop small parts of the app in cycles, allowing for adjustments along the way.
2. **Most Popular Model**:

    - Agile is widely used in the software industry because it adapts well to changing requirements and delivers working software quickly.
    - **Example**: Companies like **Spotify** and **Netflix** use Agile to continuously improve their apps and add new features based on user feedback.
3. **Fast-Paced and Flexible**:
    - Agile focuses on delivering small, functional pieces of software quickly, rather than waiting for the entire product to be completed.
    - **Example**: If you're building a **task management app**, you might release a basic version with just task creation and due dates in the first iteration, then add features like reminders and collaboration in later iterations.
4. **Enables All Phases for Everybody**:
    - In Agile, cross-functional teams (developers, testers, designers, etc.) work together throughout the project, ensuring collaboration and shared responsibility.
    - **Example**: In a **website development project**, designers, developers, and testers work together in each sprint to ensure the website is functional, user-friendly, and bug-free.
5. **Work is Broken into Small Iterations (Sprints)**:
    - Agile divides the project into small, manageable chunks called **sprints**, typically lasting 2-4 weeks. Each sprint delivers a working piece of software.
    - **Example**: If you're building an **e-commerce app**, the work might be divided into sprints like this:
        - **Sprint 1**: User registration and login.
        - **Sprint 2**: Product browsing and search.
        - **Sprint 3**: Shopping cart and checkout.
        - **Sprint 4**: Payment gateway integration.

How Agile Works in Practice:

Let's take an example of building a **fitness tracking app**:

1. **Sprint 1 (2 weeks)**:
    - Goal: Develop a basic app with user registration and login.
    - Outcome: Users can create an account and log in.
2. **Sprint 2 (2 weeks)**:
    - Goal: Add a feature to track daily steps.
    - Outcome: Users can see their step count for the day.
3. **Sprint 3 (2 weeks)**:
    - Goal: Add a feature to set fitness goals.
    - Outcome: Users can set goals (e.g., 10,000 steps per day) and track progress.
4. **Sprint 4 (2 weeks)**:
    - Goal: Add a feature to share progress with friends.
    - Outcome: Users can share their fitness achievements on social media.

Benefits of Agile:

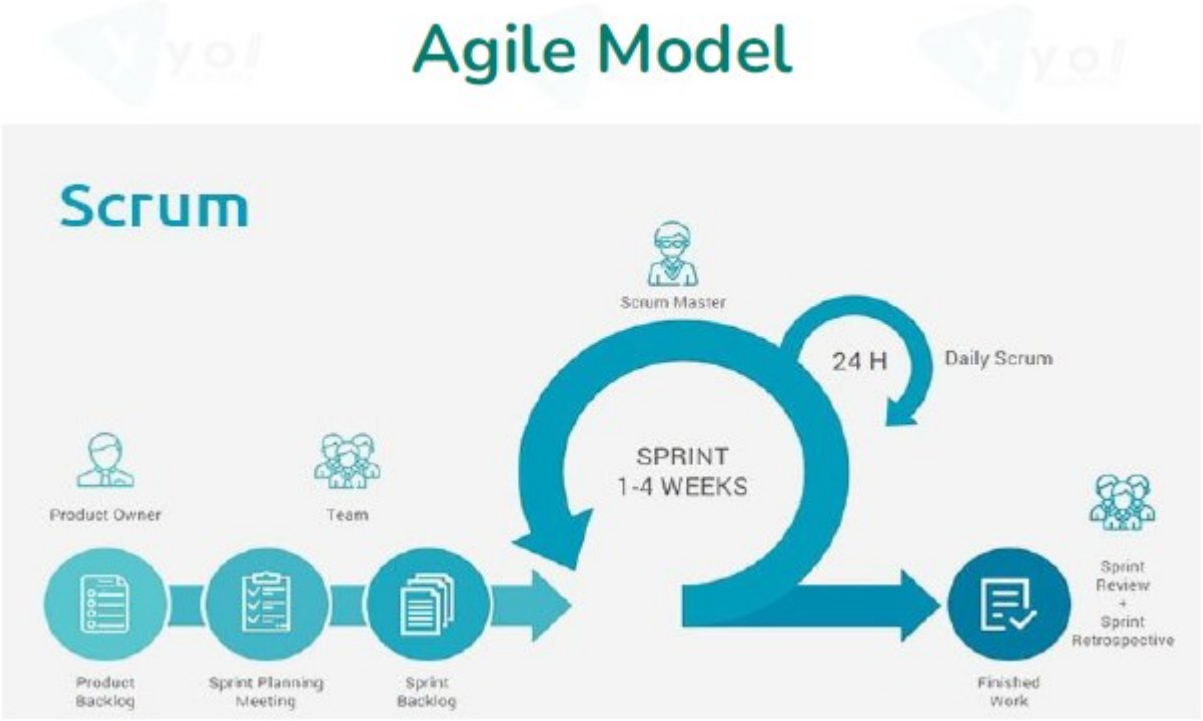
- **Flexibility**: Changes can be made easily based on feedback.

- **Faster Delivery**: Working software is delivered in small increments.
- **Collaboration**: Teams work together closely, improving communication and efficiency.
- **Customer Satisfaction**: Regular feedback ensures the product meets user needs.

Summary with Examples:

- Agile is a **flexible, iterative approach** to software development.
- Work is divided into **sprints** (e.g., 2-4 weeks), with each sprint delivering a working piece of software.
- Example: Building a **fitness tracking app** in sprints, starting with user registration and adding features like step tracking and goal setting in later sprints.

Agile is ideal for projects where requirements may change or where you want to deliver value to users quickly and continuously.



The document describes the **Agile model** with a focus on **Scrum**, one of the most popular frameworks for implementing Agile. Let's break it down with **examples** to make it easier to understand:

Key Components of Scrum:

1. **Scrum Master**:
    - **Role**: The Scrum Master is like a coach for the team, ensuring that the Scrum process is followed and helping the team overcome obstacles.

- **Example**: In a **mobile app development project**, the Scrum Master ensures that the team stays on track during sprints, removes any blockers (e.g., lack of resources), and facilitates meetings like the Daily Scrum.

2. **Daily Scrum (Daily Standup)**:
   - **What it is**: A short, daily meeting (usually 15 minutes) where team members discuss what they did yesterday, what they plan to do today, and any obstacles they're facing.
   - **Example**: In a **website development project**, the team might have a Daily Scrum where:
     - The front-end developer says, "Yesterday, I worked on the homepage design. Today, I'll start on the contact page. I'm blocked because I need the backend API to be ready."
     - The backend developer responds, "I'll have the API ready by this afternoon."
3. **Sprint**:
   - **What it is**: A time-boxed period (usually 1-4 weeks) during which the team works to complete a set of tasks from the product backlog.
   - **Example**: In a **fitness tracking app project**, a 2-week sprint might focus on developing the step-tracking feature. At the end of the sprint, the team delivers a working version of this feature.
4. **Product Owner**:
   - **Role**: The Product Owner represents the stakeholders and is responsible for prioritizing the product backlog and ensuring the team delivers value to the users.
   - **Example**: In an **e-commerce app project**, the Product Owner prioritizes features like user registration, product search, and checkout based on business goals and user needs.
5. **Team**:
   - **Role**: The cross-functional team (developers, testers, designers, etc.) works together to deliver the product increment at the end of each sprint.
   - **Example**: In a **social media app project**, the team might include:
     - Developers writing code for new features.
     - Designers creating user interfaces.
     - Testers ensuring the app is bug-free.
6. **Sprint Planning**:

- **What it is**: A meeting at the start of each sprint where the team plans the work to be done during the sprint.
- **Example**: In a **task management app project**, the team might plan to work on features like task creation, due date reminders, and task categorization during the sprint.

7. **Sprint Review**:
   - **What it is**: A meeting at the end of each sprint where the team demonstrates the completed work to stakeholders and gathers feedback.
   - **Example**: In a **music streaming app project**, the team might demonstrate a new playlist creation feature to stakeholders and gather feedback for improvements.
8. **Sprint Retrospective**:
   - **What it is**: A meeting at the end of each sprint where the team reflects on what went well, what didn't, and how to improve in the next sprint.
   - **Example**: In a **weather app project**, the team might discuss:
     - What went well: The new weather animation feature was completed on time.
     - What didn't: There were delays in integrating the API.
     - Improvements: Allocate more time for API integration in the next sprint.
9. **Product Backlog**:
   - **What it is**: A prioritized list of features, enhancements, and bug fixes that need to be addressed in the project.
   - **Example**: In a **food delivery app project**, the product backlog might include:
     - High priority: User registration and login.
     - Medium priority: Menu browsing and ordering.
     - Low priority: Loyalty rewards program.
10. **Finished Work**:
    - **What it is**: The completed tasks or features that are ready for delivery at the end of a sprint.
    - **Example**: In a **travel booking app project**, the finished work at the end of a sprint might include:
      - A working flight search feature.
      - A basic booking system.

How Scrum Works in Practice:

Let’s take an example of building a **task management app**:

1. **Sprint Planning**:
    - The team decides to work on task creation, due date reminders, and task categorization in the next 2-week sprint.
2. **Daily Scrum**:
    - Each day, the team meets for 15 minutes to discuss progress and blockers. For example:
        - Developer: “I’m working on the task creation feature. I need help with the database design.”
        - Designer: “I’ll have the UI for due date reminders ready by tomorrow.”
3. **Sprint Review**:
    - At the end of the sprint, the team demonstrates the new features to stakeholders and gathers feedback.
4. **Sprint Retrospective**:
    - The team reflects on the sprint:
        - What went well: Task creation was completed on time.
        - What didn’t: Due date reminders had some bugs.
        - Improvements: Allocate more time for testing in the next sprint.
5. **Product Backlog**:
    - The Product Owner updates the backlog, prioritizing the next set of features (e.g., task sharing and collaboration).

Summary with Examples:

- **Scrum** is a framework for implementing Agile, with time-boxed iterations called **sprints** (e.g., 2 weeks).
- Key roles include the **Scrum Master**, **Product Owner**, and **Team**.
- Key events include **Daily Scrum**, **Sprint Planning**, **Sprint Review**, and **Sprint Retrospective**.
- Example: Building a **task management app** in sprints, with each sprint delivering working features like task creation and due date reminders.

Scrum helps teams deliver high-quality software quickly and adapt to changing requirements through collaboration and continuous improvement.

What is Software Testing?

- Process that identifies the *correctness*, *completeness* and *quality*
- A *bug* is why Software Testing exists

What is a bug?

- A bug is an *unexpected result*
- Occurs when the software does not do what it's intended to do
- It's important to *identify and fix* the bug in early stages

Testers and DevOpd do not work on Production environment like : bestbuy.com, but on a previous environment like: dev-bestbuy.com, uat-bestbuy.com.

Software testing is a critical process in software development that ensures the software behaves as expected. It involves identifying the correctness, completeness, and quality of the software. The primary goal is to find and fix bugs before the software is released to users.

A **bug** is an unexpected result or behavior in the software. It occurs when the software does not perform as intended. For example, if a calculator application is designed to add two numbers but instead multiplies them, this unexpected behavior is a bug.

Here are some examples to illustrate:

1. **Correctness**: Suppose you have a login system that should allow access only if the username and password are correct. If the system grants access with an incorrect password, this is a correctness issue.
2. **Completeness**: Consider a form that requires users to fill in all fields before submission. If the form allows submission with missing information, it lacks completeness.
3. **Quality**: If a mobile app crashes frequently or has a slow response time, it indicates a quality issue, even if it functions correctly otherwise.

Identifying and fixing bugs early in the development process is crucial. For instance, if a bug in a banking application causes incorrect balance calculations, detecting and resolving it early can prevent financial discrepancies and maintain user trust.

In summary, software testing helps ensure that the software works as intended, is complete, and meets quality standards by identifying and fixing bugs early in the development cycle.

The image explains **Software Testing** and **Bugs**. Let’s break it down with detailed explanations and real-world examples.

What is Software Testing?

Software Testing is the process of verifying that a software application functions correctly, is complete, and maintains quality. It helps identify issues before releasing the product to users.

Example:
A **banking app** must be tested to ensure:

- Users can log in securely.
- Transactions are processed correctly.
- The correct balance is displayed after deposits/withdrawals.
- The app does not crash when multiple users access it at the same time.

If these aspects are not tested, the software may fail in real-world usage.

What is a Bug?

A **bug** is an error or flaw in the software that leads to an **unexpected result**. Bugs occur when the software does not perform as intended.

Example of a Bug:

- In an **e-commerce website**, a customer adds an item to the cart, but at checkout, the cart is empty. This is a bug.
- In a **messaging app**, sending an emoji causes the app to crash. This is an unexpected result and needs fixing.
- In a **ticket booking system**, selecting a date for a flight automatically changes the selected destination. This is a bug that affects user experience.

Why Fix Bugs Early?
Fixing bugs in early stages (during development and testing) saves time and money compared to fixing them after release.

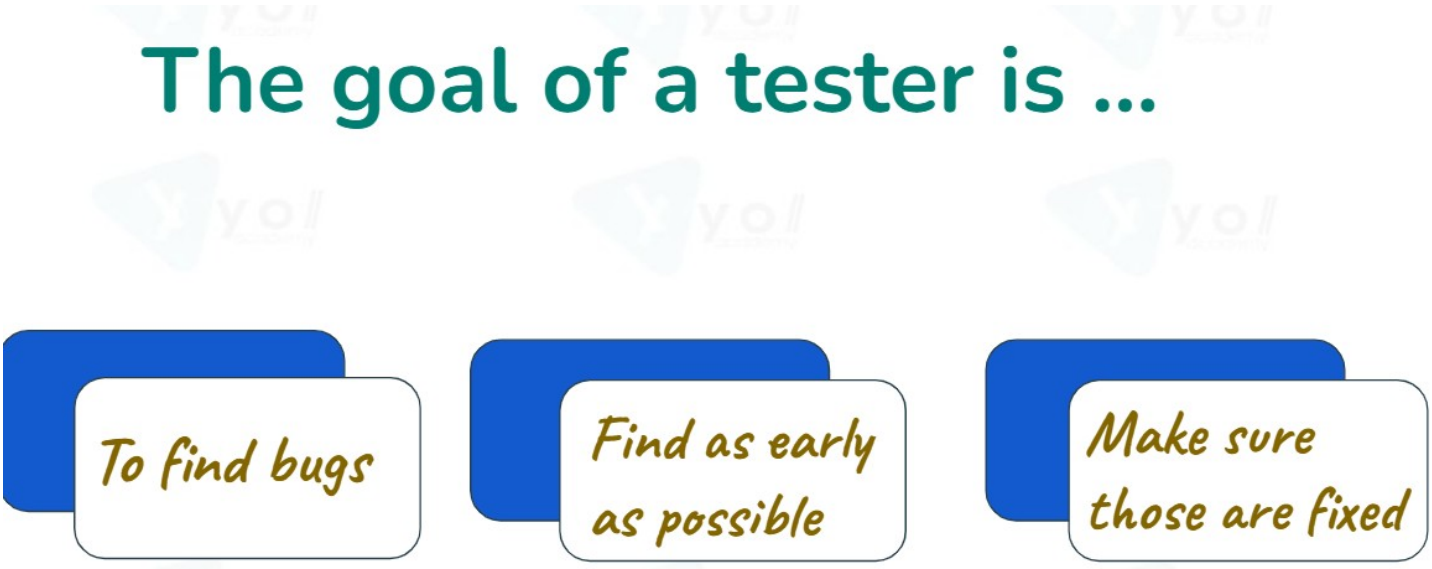
Example:

- If a **ride-sharing app** like Uber has a **payment bug** (charging double), users will stop using it, causing loss of revenue and reputation.

Why is Software Testing Important?

- Ensures that the application meets **user expectations**.
- Helps prevent **security vulnerabilities**.
- Saves **costs** by identifying issues before deployment.
- Increases **customer satisfaction** by delivering a bug-free experience.

Would you like more details on specific testing methods like **manual testing, automation testing, or performance testing**? 😊



The image explains the **goal of a software tester**, which consists of three key objectives:

---

1. To Find Bugs

A tester’s primary job is to **identify defects or issues** in the software before it reaches users.

 Example:

- A mobile banking app **fails to process payments** when transferring money between accounts. A tester must **report this bug** to developers.
- A food delivery app crashes when a user applies a promo code—this needs to be caught in testing before release.

---

2. Find as Early as Possible

Detecting bugs in the **early stages of development** reduces costs and prevents major failures.

 Example:

- If a **shopping website** has an issue in its cart functionality, identifying and fixing it during **development** is much cheaper than after launching to thousands of users.
- If a bug in **an airline booking system** causes incorrect pricing, detecting it early prevents financial losses.

---

3. Make Sure Those are Fixed

Testers must **verify that the bugs are fixed** and don’t reappear after updates.

 Example:

- After reporting a **login failure issue** in a social media app, testers **retest** it after developers fix it to ensure the login works properly.
- In a **ride-sharing app**, a tester checks if a previously fixed bug (such as incorrect fare calculation) works correctly in different locations and ride types.

Why is this important?

A good tester ensures **high software quality**, enhances **user experience**, and prevents **business losses**.

Two major categories of Software
System Software
Operating system (Windows, Mac)
Application Software
MS Word, or Web Apps like Amazon

Standalone Software

 Definition:
Standalone software is a type of application that can run independently on a computer or device **without requiring internet connectivity** or additional software dependencies. It is installed locally and does not rely on web-based or cloud services.

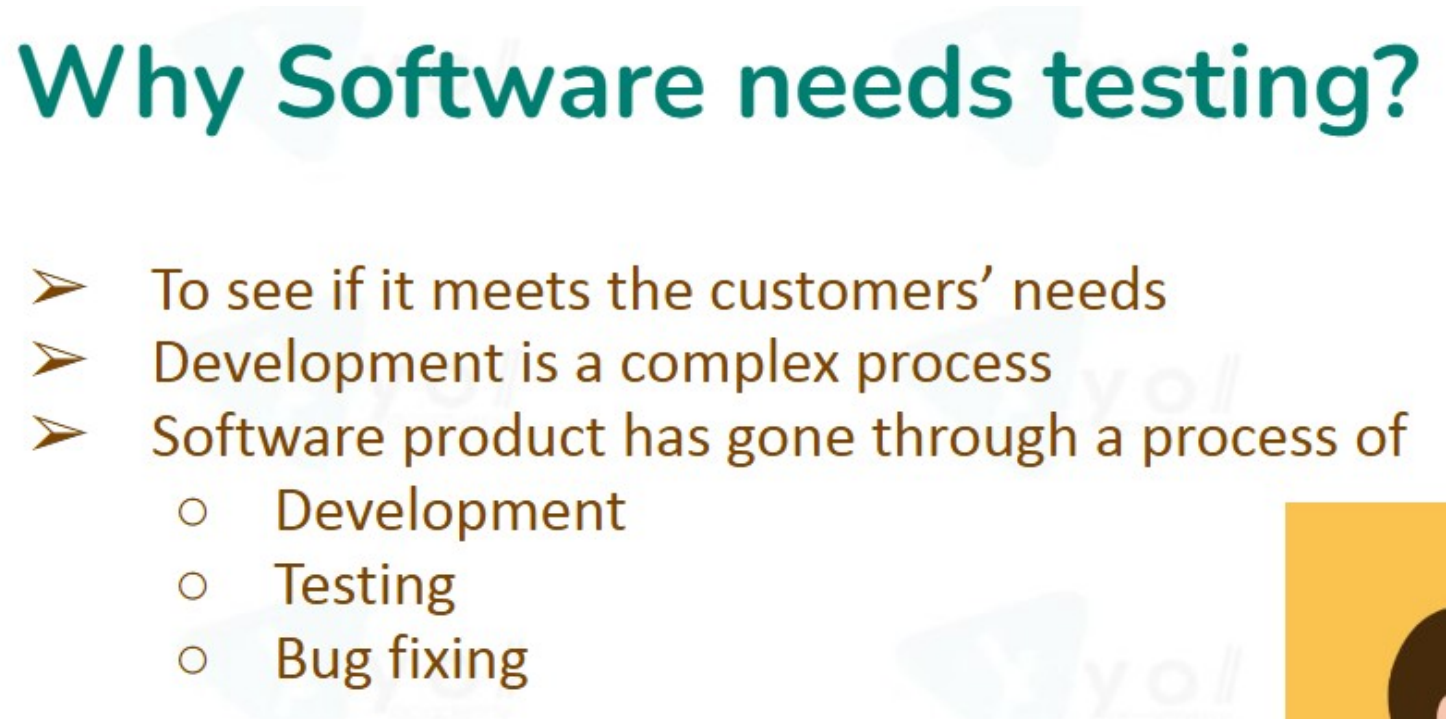
 Examples of Standalone Software:

 Microsoft Office (Offline Version) – Microsoft Word, Excel, and PowerPoint that run without internet access.

 Adobe Photoshop

□ Standalone Software vs. Web-Based Software

Feature	Standalone Software	Web-Based Software
Internet Requirement	□ Not needed	□ Needed
Installation	□ Installed on the device	□ No installation required
Performance	□ Faster (no dependency on internet speed)	⚠ Slower (depends on server response)
Examples	MS Word (Offline), Photoshop, VLC	Google Docs, Canva, YouTube

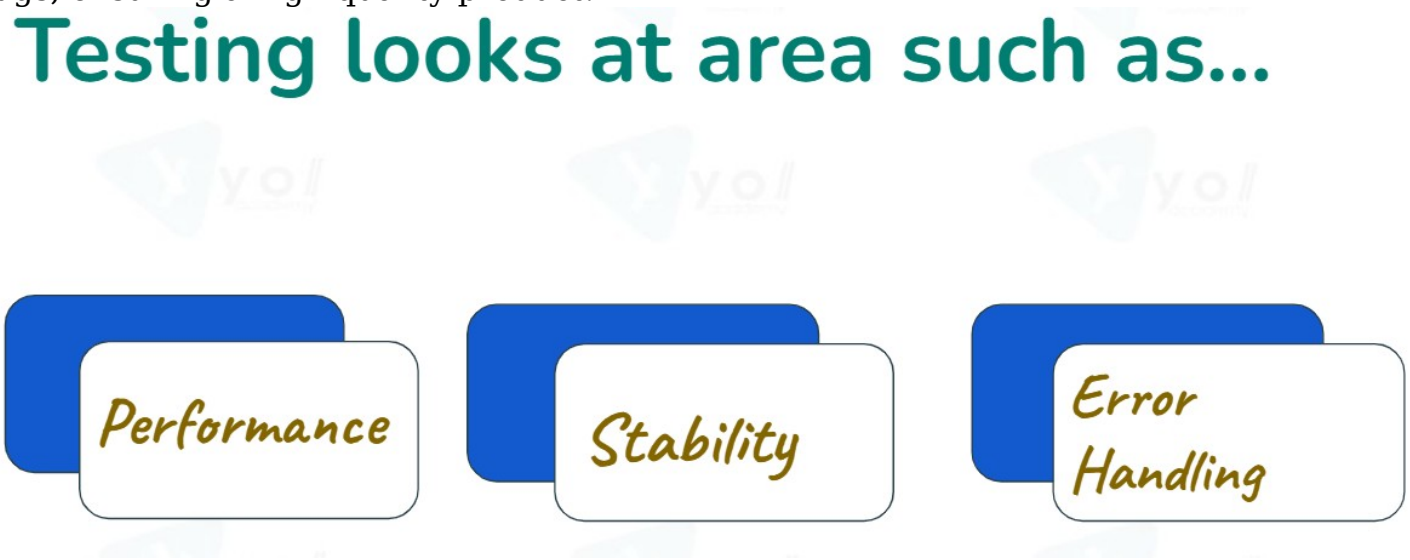

Why Software needs testing?
To see if it meets the customers' needs
Development is a complex process
Software product has gone through a process of
Development
Testing
Bug fixing

Software testing is essential to ensure that the final product meets the customers' needs and functions as intended. The development of software is a complex process involving multiple stages, including development, testing, and bug fixing. Here's why software needs testing, explained with examples:

1. **Meeting Customer Needs**: Software is created to solve specific problems or fulfill particular needs for users. Testing ensures that the software meets these requirements. For example, if a customer needs a mobile app to track daily expenses, testing verifies that the app accurately records and categorizes expenses as expected.
2. **Complex Development Process**: Software development involves many components and interactions that can lead to errors. Testing helps identify these issues. For instance, in a large e-commerce platform, testing ensures that the shopping cart, payment gateway, and user authentication systems work seamlessly together.

3. **Bug Identification and Fixing**: During development, bugs can be introduced. Testing helps identify these bugs so they can be fixed before the software is released. For example, if a social media app has a bug that prevents users from uploading photos, testing would catch this issue, allowing developers to fix it before the app goes live.
4. **Ensuring Quality**: Testing ensures the software is reliable, user-friendly, and performs well under various conditions. For example, testing a video streaming service would involve checking if it can handle high traffic without crashing and if the video quality remains consistent.

In summary, software testing is crucial to verify that the software meets customer needs, functions correctly despite the complexity of development, and is free of critical bugs, ensuring a high-quality product.



The image outlines three key areas of software testing:

1. **Performance**
   - This refers to how well a system responds under different conditions, including load and stress testing.
   - **Example:** Checking how fast a website loads when 1,000 users try to access it simultaneously.
2. **Stability**
   - Stability testing ensures that the application runs smoothly over time without unexpected crashes or failures.
   - **Example:** Running an e-commerce application continuously for 48 hours to see if it crashes or slows down.
3. **Error Handling**
   - This involves verifying that the system correctly manages errors and provides appropriate feedback to users.
   - **Example:** If a user enters an incorrect password multiple times, the system should display a message like "Incorrect password, try again" instead of crashing.

These areas help ensure a reliable, efficient, and user-friendly software experience.

The image explains the concept of quality and its importance in software or product development.

What is Quality?

Quality is defined as a **"degree of excellence"**, which means delivering a product or service that meets or exceeds customer expectations. Key aspects include:

- **Exceeding customer expectations**: Providing more value than the customer anticipates.
- **Good performance on platforms**: Ensuring the product works efficiently across various platforms (web, mobile, desktop).
- **No negative impact on end users**: Ensuring the software does not cause inconvenience, crashes, or security issues.

Why is Quality Needed?

Quality is crucial for business success because:

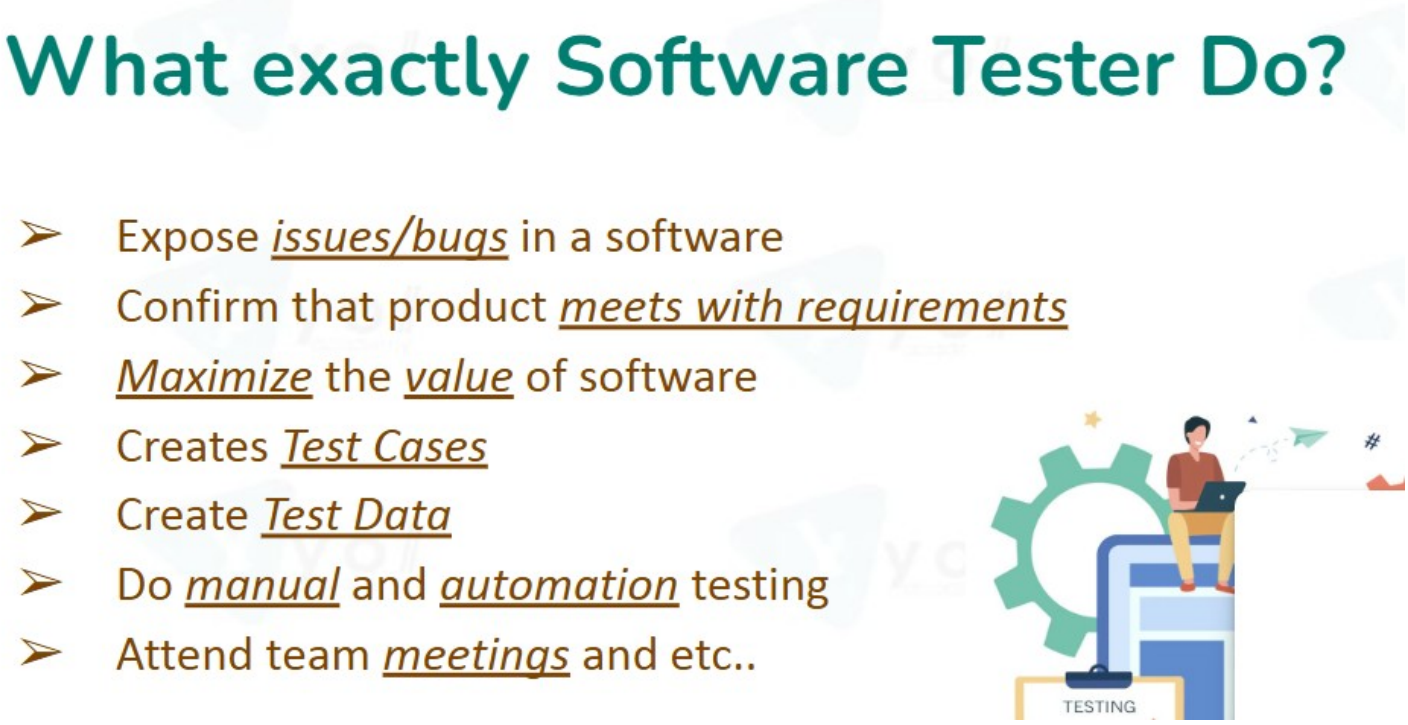
- **Saves time and money**: Fixing defects early prevents costly rework.
- **Increases customer satisfaction**: Users prefer products that work seamlessly.
- **Helps to get more business**: A high-quality product attracts new customers and retains existing ones.

Example in Software Development:

A **high-quality e-commerce website** will:

- Load quickly and handle high traffic without slowing down.
- Provide a seamless checkout process without errors.
- Ensure secure transactions to protect user data.

This demonstrates why quality is essential for both customer experience and business growth.


What exactly Software Tester Do?
Expose issues/bugs in a software
Confirm that product meets with requirements
Maximize the value of software
Creates Test Cases
Create Test Data
Do manual and automation testing
Attend team meetings and etc..
TESTING

The image describes the responsibilities of a **Software Tester**, who plays a crucial role in ensuring the quality and reliability of software. Here's a detailed explanation with examples:

1. Expose issues/bugs in a software

- Testers identify and report defects, errors, or unexpected behavior in the application.
- **Example:** If an online shopping app crashes when a user applies a discount code, the tester reports this as a bug.

2. Confirm that product meets requirements

- Ensure that the software functions as per the specified business and technical requirements.
- **Example:** A banking app should allow users to transfer money. If the "Transfer" button does nothing, it fails to meet the requirement.

3. Maximize the value of software

- Ensure the software is user-friendly, efficient, and delivers a seamless experience.
- **Example:** A tester may suggest improving the navigation of a travel booking website to help users find flights faster.

4. Create Test Cases

- Write step-by-step scenarios to verify different functionalities of the software.
- **Example:** A test case for a login page might include entering a valid username and password, clicking "Login," and checking if the user is redirected to the dashboard.

5. Create Test Data

- Generate data sets to simulate real-world usage.
- **Example:** For an insurance software, a tester might create test data with different age groups and health conditions to ensure correct premium calculations.

6. Perform Manual and Automation Testing

- **Manual Testing:** Testers manually check software without using automation tools.
- **Automation Testing:** Using tools like Selenium or JUnit to automate repetitive test cases.
- **Example:** Running an automated test to verify that a website loads correctly on different browsers.

7. Attend team meetings

- Discuss testing progress, report issues, and collaborate with developers and business teams.
- **Example:** In an Agile team, testers participate in daily stand-up meetings to share updates on testing activities.

By performing these tasks, software testers **ensure software reliability, enhance user experience, and prevent costly errors before release**.

Types of Test Data

1 Valid Test Data

- **Definition**: Data that meets the application's expected input criteria.
- **Example**:
  - For a login form:
    - **Username:** john_doe
    - **Password:** Pass@123
  - Expected behavior: Successful login.

2 Invalid Test Data

- **Definition**: Data that does not meet the application's expected input criteria.

- **Example**:
  - Username: johndoe!@# (special characters not allowed)
  - Expected behavior: Show an error message.

3 Boundary Test Data

- **Definition**: Data that tests the boundary limits of input fields.
- **Example**:
  - A password field accepts **8-16 characters**:
    - □ Pass1 (Too short) → Should fail
    - □ PassWord@123456789 (Too long) → Should fail
    - □ Pass@123 (Valid length) → Should pass

4 Null or Empty Test Data

- **Definition**: Testing how the application handles empty input.
- **Example**:
  - Leaving the **email field blank** in a registration form.
  - Expected behavior: Show "Email is required" message.

5 SQL Injection Test Data

- **Definition**: Data used to check if the system is vulnerable to SQL Injection attacks.
- **Example**:
  - Entering "' OR '1'='1' --" in a login field.
  - Expected behavior: The system **should block this input**.

6 Large Data Volume Test

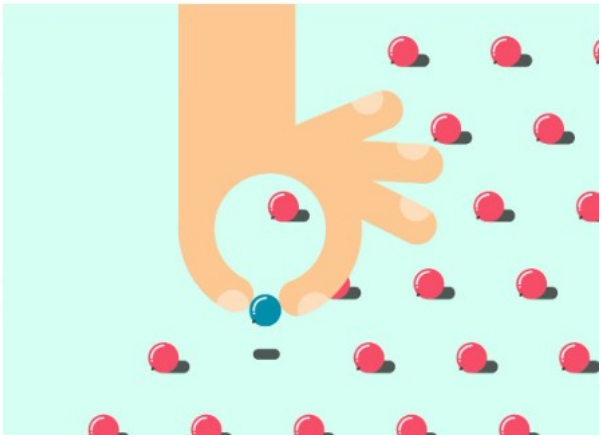
- **Definition**: Entering a large amount of data to check performance.
- **Example**:
  - Uploading a **5GB file** to test a file upload system.
  - Expected behavior: System should handle it properly.

□ Why is Test Data Important?

□ Ensures all application scenarios are tested.
□ Helps find bugs related to **data validation, security, and performance**.
□ Improves the **reliability** of the software before release.

What are the Traits of a Good Tester?

- Know the Technology
- Know how to troubleshoot
- Perfectionist and Realist
- Tactful, diplomatic, persuasive
- Explorer
- Organized
- Objective and Accurate
- Learn from Defects



Traits of a Good

A good software tester possesses several qualities that help ensure the software is **reliable, bug-free, and meets user expectations**. Below are the key traits with practical examples:

---

1 Know the Technology

A tester should understand the technology behind the software being tested.

Example:

- If testing a web application, the tester should know about **HTML, CSS, JavaScript, APIs, and databases**.
- When testing a mobile app, understanding **Android/iOS environments** is helpful.

---

2 Know How to Troubleshoot

Testers should have problem-solving skills to investigate and reproduce bugs.

Example:

- If a **login feature** is failing, the tester should check **logs, network requests, and backend responses** to find the cause.
- When an app **crashes on iOS but works fine on Android**, the tester should analyze device compatibility issues.

---

3 Perfectionist and Realist

A good tester aims for perfection but also understands **deadlines and constraints**.

Example:

- A tester finds **5 minor UI issues** before launch but understands that fixing them might **delay the release**. They prioritize the most critical ones.

---

4 Tactful, Diplomatic, Persuasive

Testers must communicate bugs **without offending developers** and **convince stakeholders** of their importance.

Example:

- Instead of saying, *"Your code is broken,"* a tester should say, *"I found an issue in this feature, here's how it can be improved."*
- If a manager wants to **release software with critical bugs**, the tester should provide **data and impact analysis** to persuade them to reconsider.

---

5 Explorer (Curious and Creative Thinking)

Testers should think outside the box to find unexpected bugs.

Example:

- A tester types **emoji characters in a name field** to see if it crashes the system.
- They **input negative numbers in a payment field** to test validation.

---

6 Organized

A tester should maintain **test cases, bug reports, and documentation** properly.

Example:

- Using **JIRA, TestRail, or Excel** to track testing progress.
- Keeping **detailed bug reports** with **steps to reproduce, screenshots, and logs**.

---

7 Objective and Accurate

A tester must report bugs based on facts, not opinions.

Example:

- Instead of saying, *"This feature is bad,"* they should provide **evidence** like, *"On clicking the 'Submit' button, nothing happens. Expected behavior: It should save the form."*

---

8 Learn from Defects

A good tester **analyzes past bugs** to prevent them in the future.

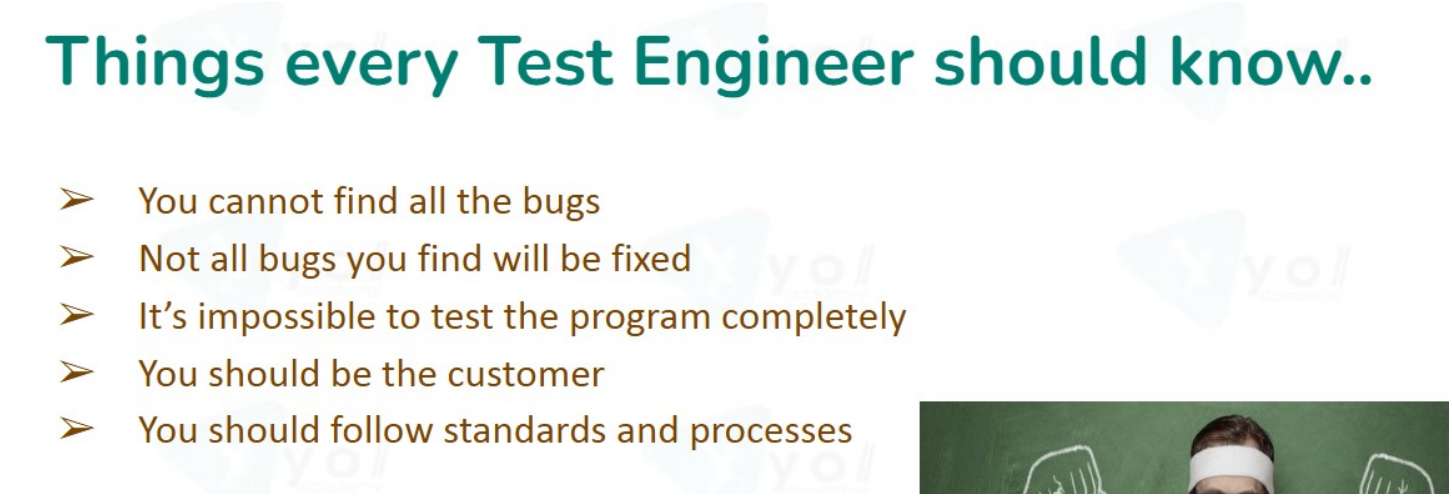
Example:

- If a previous release had **security vulnerabilities**, the tester ensures **security testing is a priority** for upcoming versions.
- If a mobile app previously failed due to **memory leaks**, they include **performance testing** in future cycles.

Summary

A great tester is a mix of **technical expert, detective, and diplomat**. They **find, analyze, and report defects** while ensuring **smooth collaboration** with developers and product teams.

Would you like **real-world bug examples** or guidance on **test case writing**? 😊


Things every Test Engineer should know..
You cannot find all the bugs
Not all bugs you find will be fixed
It's impossible to test the program completely
You should be the customer
You should follow standards and processes

Manual Testing Demo

Sample Test Case

Test Case Description:

User can login to the MealB app and can **add expense** to “Meal & Entertainment” expense.

Instructions:

Step 1 - Go to MealB URL and enter login credentials to login to the MealB Application

Step 2 - Once you are on the MealB Application hover over on ***+ (plus)*** sign on the right hand side

Step 3 - Click on the “Meal & Entertainment” button

Step 4 - Fill out the required fields and click on “Save” button

Step 5 - Validate that the expense was created

Expected Result:

Step 1 - Tester should be able to enter credentials and login

Step 2- Tester should be able to hover over the plus button and see “Meal & Entertainment” expense option

Step 3 - Tester should be able to enter details in required fields

Step 4 - Tester should be able to save details

Step 5 - Tester should be able to see the expense created on the expenses list

Actual Result:

Actual Results:

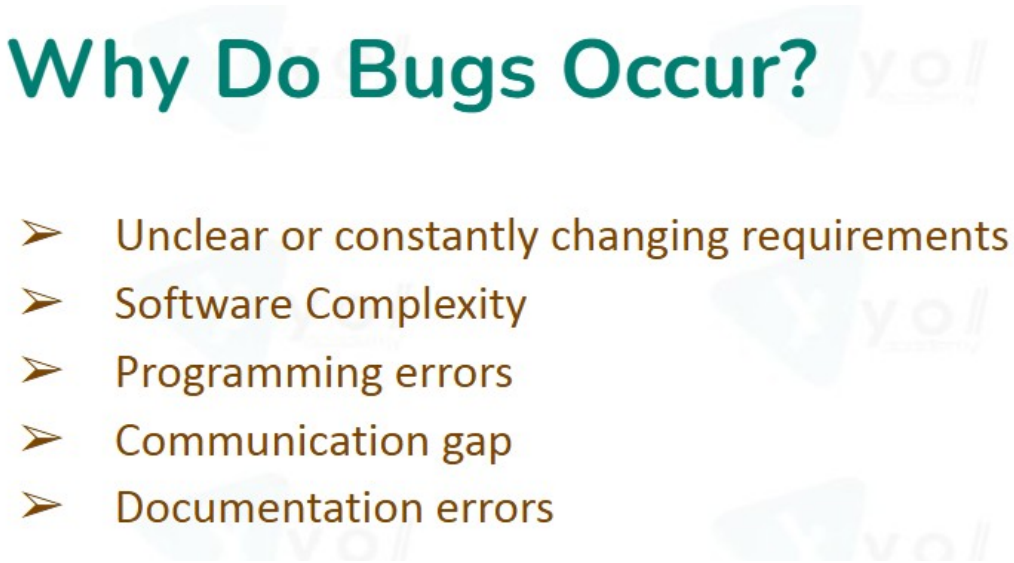
Step 1 - Tester is able to input credentials and login

Step 2 - Tester is able to hover over on the plus button and see “Meal & Entertainment” expense option

Step 3 - Tester is able to add expense

Step 4 - Tester is able to add details

Step 5 - Tester sees the expense created



Bugs occur in software due to various reasons, ranging from unclear requirements to human errors. Let's go through each cause with examples.

---

1 Unclear or Constantly Changing Requirements

What Happens?

- Developers may build features based on initial requirements, but **if requirements change frequently**, it leads to mismatches between expectations and implementation.
- Testers may not get **updated test cases**, leading to undetected defects.

Example:

- A client initially requests a **login page with only email and password**, but later changes it to **support login via mobile number and social media**.
- If developers and testers aren’t informed properly, some login methods may **not work correctly**, leading to a **buggy user experience**.

---

2 Software Complexity

What Happens?

- Large-scale applications have multiple **modules, dependencies, and integrations**, making them difficult to maintain.
- A small change in one module may **break another module**.

Example:

- In an **e-commerce website**, the **shopping cart** is linked to **inventory management**.
- If the inventory team updates stock management logic, it may cause issues in the cart, leading to **wrong stock availability displayed**.

---

3 Programming Errors

□ What Happens?

- Developers make **mistakes while writing code**, leading to logic errors, syntax errors, or incorrect calculations.

□ Example:

- A developer mistakenly writes:

python

total_price = item_price - discount

Instead of:

python

total_price = item_price - (item_price * discount / 100)

- This results in **wrong discount calculations**, affecting all purchases.

---

4 Communication Gap

□ What Happens?

- If **business analysts, developers, and testers** do not communicate clearly, misunderstandings can occur.
- Developers may interpret requirements differently from what testers or clients expect.

□ Example:

- A requirement states:
  "Users should be able to update their email"
- The developer **only allows changing email but does not verify it**.
- The tester **assumes verification is required**, but since it wasn’t communicated properly, the bug is missed.

---

5 Documentation Errors

What Happens?

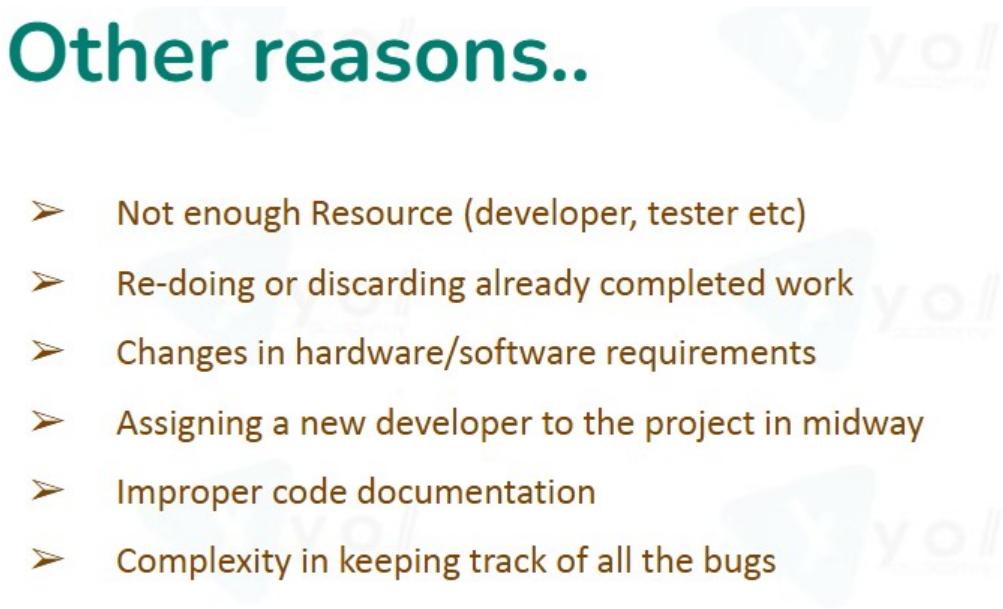
- If documentation such as **SRS (Software Requirement Specification), user stories, or test cases** has incorrect or missing details, it can lead to defects.

Example:

- The **requirement document** states:
  - "The mobile number field should accept only **10 digits**."
- But the developer mistakenly assumes **international numbers should be allowed**, causing **validation errors** in different countries.

Conclusion

Bugs are inevitable in software development, but **clear communication, strong documentation, proper testing, and better requirement management** can reduce them.


Other reasons..
Not enough Resource (developer, tester etc)
Re-doing or discarding already completed work
Changes in hardware/software requirements
Assigning a new developer to the project in midway
Improper code documentation
Complexity in keeping track of all the bugs

Bugs can occur due to various factors beyond coding errors and unclear requirements. Here are some additional reasons why bugs may appear in software development, along with real-world examples.

---

1 Not Enough Resources (Developers, Testers, etc.)

What Happens?

- If there aren't **enough developers, testers, or project managers**, workloads increase, leading to **hurried coding and testing**.

- Testers may **skip** certain test scenarios due to tight deadlines.

Example:

- A **startup company** develops an e-commerce website with only **two developers and one tester**.
- Due to time constraints, the tester focuses on **functional testing** but skips **security testing**.
- Later, users find that **payment details are exposed**, leading to a security breach.

---

2 Re-doing or Discarding Already Completed Work

What Happens?

- When a project undergoes **frequent redesigns or changes**, developers must **rewrite existing code**.
- This increases the risk of introducing **new defects** while modifying old functionality.

Example:

- A banking app initially **supports only local transactions**, but mid-project, the client requests **international payments**.
- Developers must modify the **entire payment module**, leading to **unexpected bugs** in transaction processing.

---

3 Changes in Hardware/Software Requirements

What Happens?

- If the software is designed for **specific hardware or operating system** and later has to support **different configurations**, compatibility issues arise.

Example:

- A **mobile game** is developed for **Android 11+**, but the company later decides to **support Android 8+**.
- The game crashes on older devices because they lack **the latest GPU optimizations**.

---

4 Assigning a New Developer to the Project in Midway

What Happens?

- When a **new developer joins mid-project**, they may not fully understand the existing **code structure and logic**.
- If documentation is poor, they might **misinterpret functionalities** and introduce defects.

Example:

- In a **travel booking app**, a new developer joins midway and updates the **cancellation policy module**.
- Due to a misunderstanding, **users get refunds even for non-refundable tickets**, causing financial loss to the company.

---

5 Improper Code Documentation

What Happens?

- Poorly documented code makes it difficult for developers to **understand logic, dependencies, and functions**.
- Without proper comments, future developers may **modify the code incorrectly**, leading to defects.

Example:

- A **finance software** calculates **loan interest** using a formula, but the original developer **did not document** how the formula works.
- A new developer changes the formula incorrectly, leading to **wrong interest calculations** for customers.

---

6 Complexity in Keeping Track of All the Bugs

What Happens?

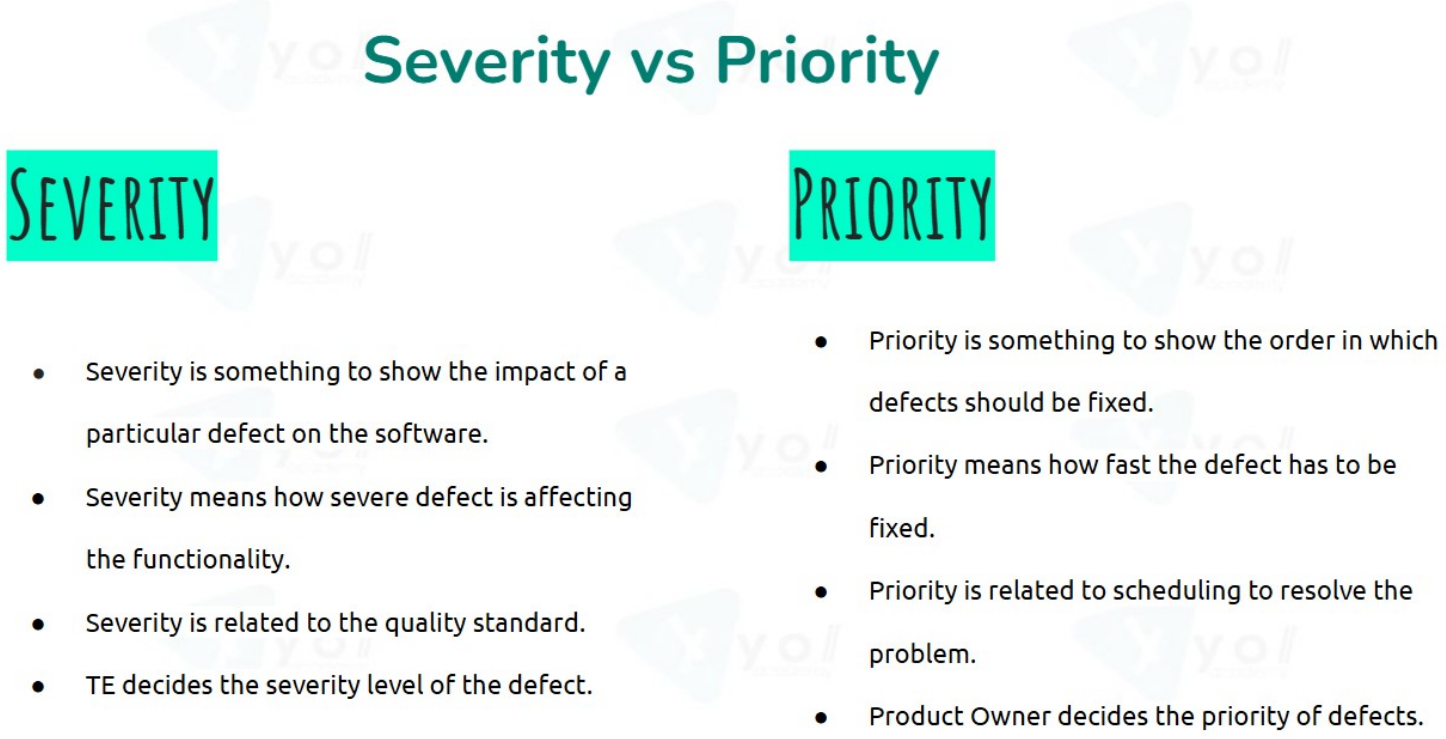
- In large projects, there may be **hundreds of bugs**, and without a proper bug-tracking system, some may be **missed or reappear**.

Example:

- A **hospital management system** has multiple bug reports.
- Due to **poor bug tracking**, a **critical bug in patient record updates** is forgotten and resurfaces after release, causing **incorrect patient reports**.

Conclusion

Bugs are not just caused by coding mistakes but also by **resource issues, documentation gaps, requirement changes, and tracking failures**. Using **agile methodologies, proper documentation, and structured bug tracking tools** can help reduce these problems.


Severity vs Priority
SEVERITY
Severity is something to show the impact of a particular defect on the software.
Severity means how severe defect is affecting the functionality.
Severity is related to the quality standard.
TE decides the severity level of the defect.
PRIORITY
Priority is something to show the order in which defects should be fixed.
Priority means how fast the defect has to be fixed.
Priority is related to scheduling to resolve the problem.
Product Owner decides the priority of defects.

Severity vs. Priority in Software Testing (With Examples)

When identifying software defects, **severity** and **priority** are key factors in deciding how to handle and resolve issues. Let's break them down with **real-world examples**.

---

1 What is Severity?

- **Severity** measures the **impact** of a defect on software functionality.
- It indicates how much **damage** the defect causes to the system.
- **Decided by: Test Engineers (TE).**

Example Scenarios:

Severity Level	Description	Example
Critical	The defect **completely blocks** system functionality.	A banking app **crashes** when trying to make a transaction.
High	A **major function** is broken but does not crash the system.	A user cannot **reset their password**, causing login issues.
Medium	A **minor feature** is malfunctioning	A shopping cart **fails to update**

Severity Level	Description	Example
	but has workarounds.	**quantities** but checkout still works.
Low	A **UI issue or minor bug** that does not affect functionality.	A **spelling mistake** in a settings page.

2 What is Priority?

- **Priority** determines **when** a defect should be fixed.
- It depends on business needs, user experience, and release deadlines.
- **Decided by: Product Owner/Project Manager.**

Example Scenarios:

Priority Level	Description	Example
High	Needs an **immediate fix** as it affects many users.	**Checkout button not working** in an e-commerce website.
Medium	Should be fixed soon, but it's **not urgent**.	A **profile picture upload fails**, but users can still use the app.
Low	Can be fixed **later**, does not impact functionality.	A **misaligned logo** in a mobile app.

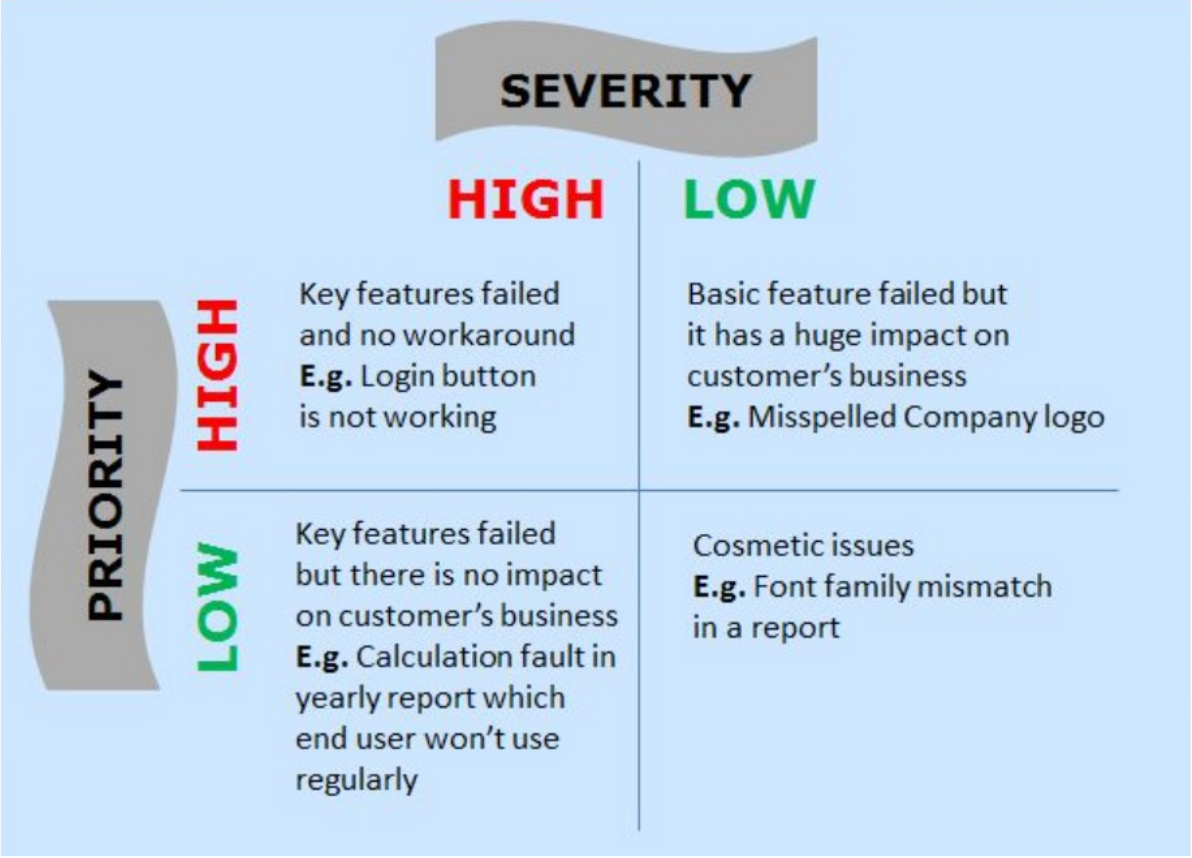
3 How Severity and Priority Work Together

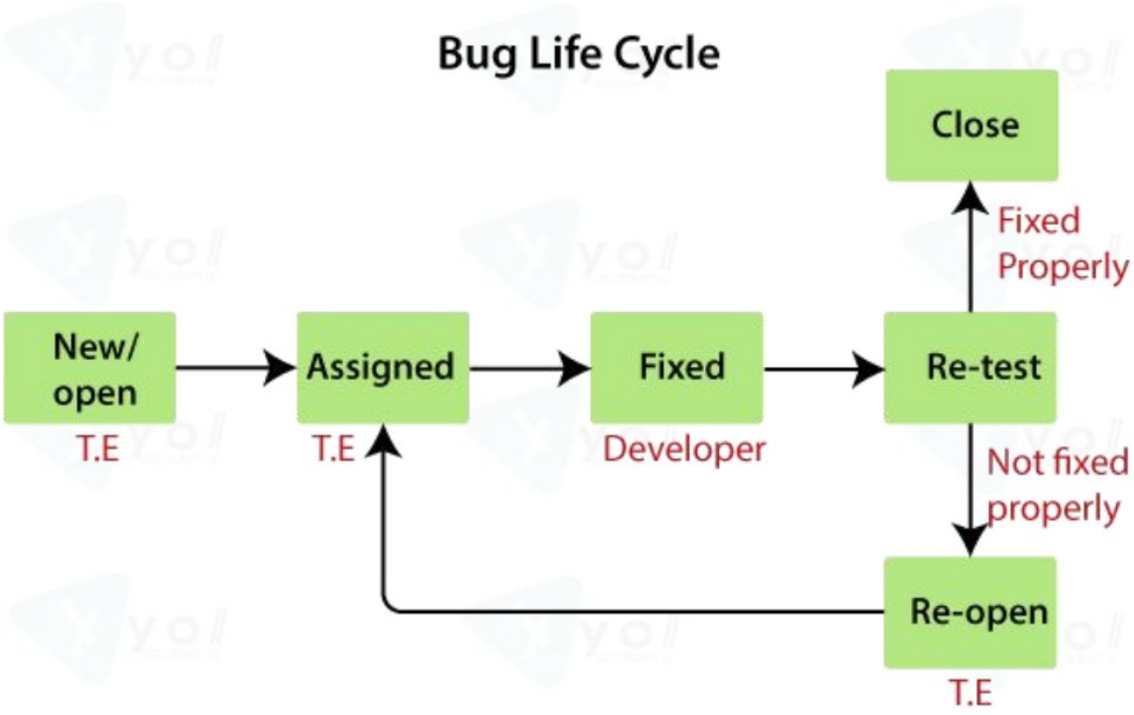
Examples of Different Combinations

Severity	Priority	Example
High Severity, High Priority	Needs an urgent fix.	**Payment system is broken**, blocking transactions.
High Severity, Low Priority	Can be fixed later.	**Admin panel crashes**, but it’s used only by internal staff.
Low Severity, High Priority	Fix quickly for user experience.	**A typo on the homepage** of an important website.
Low Severity, Low Priority	Least urgent issue.	**An outdated logo** on a rarely used settings page.

Conclusion

- **Severity** relates to the **technical impact** of the bug.
- **Priority** determines **how soon** it should be fixed.
- A **critical defect** (e.g., system crash) is usually **high severity & high priority**, but a **UI issue** may have **low severity & high priority** if it affects branding.


SEVERITY
HIGH
LOW
PRIORITY
HIGH
LOW
Key features failed and no workaround E.g. Login button is not working
Basic feature failed but it has a huge impact on customer's business E.g. Misspelled Company logo
Key features failed but there is no impact on customer's business E.g. Calculation fault in yearly report which end user won't use regularly
Cosmetic issues E.g. Font family mismatch in a report



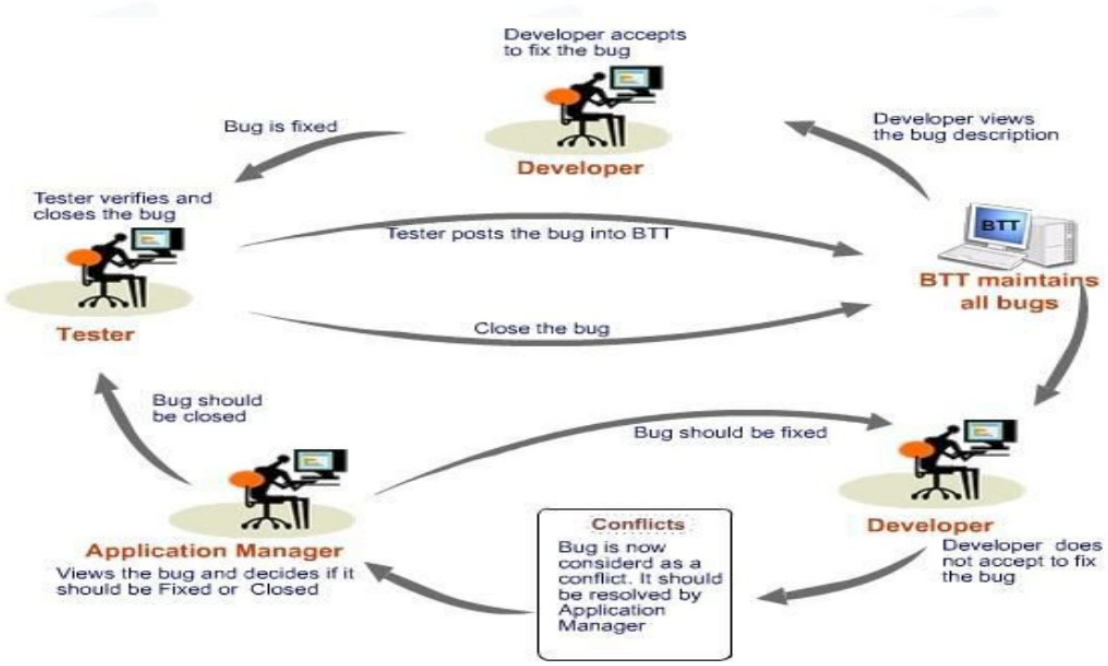
The image depicts the **Bug Life Cycle**, which represents the different states a bug goes through during software development and testing.

Bug Life Cycle Stages with Examples:

1. **New/Open:**
   - A tester Engineer (T.E) identifies a bug and logs it in a bug-tracking system.
   - Example: A login button is unresponsive when clicked.
2. **Assigned:**
   - The bug is assigned to a developer for fixing.
   - Example: The project manager assigns the bug to a frontend developer.
3. **Fixed:**
   - The developer analyzes the issue and implements a fix.
   - Example: The developer updates the button’s JavaScript event listener to handle clicks correctly.
4. **Re-test:**
   - The tester verifies whether the fix is working as expected.
   - Example: The tester tries clicking the login button again.
5. **Close (if fixed properly):**
   - If the bug is no longer present, it is marked as "Closed."
   - Example: The login button now works fine, so the tester closes the bug.

6. **Re-open (if not fixed properly):**
    - If the issue still persists, the bug is re-opened and assigned back to the developer.
    - Example: The button works in Chrome but still does not respond in Firefox. The tester reopens the bug.

This cycle continues until the bug is completely resolved and verified.


Developer accepts to fix the bug
Developer
Bug is fixed
Developer views the bug description
Tester verifies and closes the bug
Tester posts the bug into BTT
BTT
BTT maintains all bugs
Tester
Close the bug
Bug should be closed
Bug should be fixed
Application Manager
Views the bug and decides if it should be Fixed or Closed
Conflicts
Bug is now considerd as a conflict. It should be resolved by Application Manager
Developer
Developer does not accept to fix the bug

The image represents a **Bug Life Cycle** with interactions between different roles: **Tester, Developer, Application Manager, and BTT (Bug Tracking Tool)**. Here's a step-by-step breakdown with examples:

1. Tester Identifies the Bug

- A tester finds an issue in the application and logs it in **BTT (Bug Tracking Tool)**.
- Example: The search bar in an e-commerce app does not return results when a user types a product name.

2. Developer Views the Bug

- The developer checks the bug details in BTT.
- Example: The developer reviews the logs and confirms that there is an issue with the database query.

3. Developer Accepts or Rejects the Bug

- If the developer **accepts** the bug, they proceed with fixing it.
- If the developer **rejects** the bug (e.g., if it's not reproducible or a misunderstanding), a **conflict** arises.

Example of Rejection:

- The developer tests the search bar and finds that it works when the database is refreshed.
- The bug is considered a **false alarm**, and the developer rejects it.

4. Conflict Resolution by Application Manager

- If the tester still believes the issue exists, the **Application Manager** steps in to decide.
- Example: The Application Manager tests it on multiple devices and finds that the issue only occurs on mobile. They instruct the developer to fix it.

5. Developer Fixes the Bug

- The developer identifies the problem and implements a fix.
- Example: The developer updates the database query to handle mobile search requests correctly.

6. Tester Verifies the Fix

- The tester retests the feature to ensure it's working as expected.
- Example: The tester enters product names in the search bar on different devices and verifies that results appear correctly.

7. Closing the Bug

- If the fix works, the tester **closes the bug** in BTT.
- Example: The search functionality works fine across all platforms, so the tester marks the bug as "Closed."

If the bug is **not fixed properly**, the tester **reopens the bug**, and the cycle repeats until the issue is resolved.

This structured process ensures that all bugs are tracked, verified, and resolved efficiently.

Famous Bug Tracking Tools

1. Jira
2. HP Alm
3. Bugzilla
4. Backlog
5. ReQTest